MÓJ MIKRO-KOMPUTER

ZX Spectrum

Roland Wacławek

Młodzieżowa Agencja Wydawnicza

MÓJ MIKROKOMPUTER ZX Spectrum

Roland Wacławek

Młodzieżowa Agencja Wydawnicza 1987

Projekt okładki: Jarosław Jasiński
Zdjęcie na okładce: Grzegorz Dymarski
Redaktor merytoryczny: dr inż. Marek Siwiński
Redaktor: Krystyna Liszyk
Redaktor techniczny: Mirosława Bokus
Korektor: Maria Czempińska

ISBN 83-203-2595-1

RSW „Prasa-Książka-Ruch"
Młodzieżowa Agencja Wydawnicza
Warszawa 1987 r. Wydanie I
Nakład 59 700+300 egz.
Ark. wyd. 10,1. Ark. druk. 9/16
Oddano do produkcji w sierpniu 1986 r
Podpisano do druku w październiku 1987 r.
Prasowe Zakłady Graficzne
Koszalin, ul. A. Lampego 18/20
Zam. D-305 K-5

Wstęp

Mikrokomputer domowy Sinclair ZX Spectrum pojawił się na rynku w Wielkiej Brytanii w kwietniu 1982 roku. W niedługim czasie zyskał sobie licznych zwolenników nie tylko w Anglii, ale i w wielu innych krajach Europy. Przez długi czas był ZX Spectrum zdecydowanie najtańszym komputerem domowym. Wraz z Commodore C-64 odegrał istotną rolę w szybkiej obniżce cen mikrokomputerów.

Mimo prostoty dysponuje Spectrum atrakcyjnymi właściwościami: niezłą rozdzielczością graficzną oraz wygodnym, łatwym w opanowaniu językiem BASIC. ZX Spectrum nie wymaga dodatkowego, specjalistycznego osprzętu. Współpracuje z dowolnym odbiornikiem telewizyjnym jako monitorem, w charakterze pamięci masowej wykorzystuje zwykły, amatorski magnetofon. Spectrum jest sam w sobie kompletnym systemem, czego nie można powiedzieć np. o obiektywnie znacznie lepszym C-64. Nic dziwnego, że w Polsce ZX Spectrum stał się najpopularniejszym komputerem amatorskim. Trudno nie wspomnieć też o jego licznych, poważnych zastosowaniach w szkołach, placówkach naukowych, biurach i przedsiębiorstwach produkcyjnych.

Komputer domowy nie może zastąpić w pełni profesjonalnego komputera osobistego. Przeszkodą będzie niewygodna klawiatura, zbyt mała pojemność ekranu w trybie tekstowym oraz brak szybkich pamięci zewnętrznych o dostępie swobodnym — np. dysków elastycznych lub dysków tzw. twardych. Nieprawdziwe jest jednak twierdzenie, że komputery domowe nadają się wyłącznie do gier. Wykorzystane w sposób przemyślany mogą w wielu zastosowaniach praktycznych oddać poważne usługi. Zresztą nawet zabawa, byle kształcąca i twórcza, niesie poza rozrywką wiele pożytków.

Niniejsza książka przeznaczona jest głównie dla użytkowników ZX Spectrum. Nie jest ona bynajmniej zbiorem gotowych programów, lecz swego rodzaju "książką kucharską". Zawiera praktyczne wskazówki, szkice rozwiązań i przykłady oraz trochę narzędzi do samodzielnego wykorzystania we własnych programach Czytelników. Książka adresowana jest do osób zainteresowanych twórczym wykorzystaniem mikrokomputera i samodzielnym opracowywaniem programów. Z tego powodu zawiera też nieco informacji o wewnętrznym życiu Spectrum. Wiedza taka jest niezbędna do lepszego zrozumienia działania sprzętu i pozwala lepiej wykorzystać jego zalety.

Programowanie mikrokomputerów to dziedzina jak najbardziej praktyczna. Najlepiej zgłębiać ją eksperymentując przy klawiaturze. Nieudanym programem uszkodzić mikrokomputera się nie da. Co najwyżej wyświetlony zostanie komunikat o błędzie. W razie niepowodzeń nie należy jednak odsuwać maszyny ze zniecierpliwieniem.

Przyczyną problemów jest zawsze nasza niedostateczna wiedza. Błąd trzeba wytropić. Każdy samodzielnie zlokalizowany błąd powiększa naszą wiedzę o sprzęcie i sprawność w wykorzystaniu naszego nowego narzędzia: mikrokomputera.
Niniejsza książka nie jest systematycznym kursem programowania w języku BASIC. Tematowi temu poświęcone są cykle artykułów w czasopismach oraz specjalne podręczniki. Znajomość elementarnych zasad programowania w języku BASIC byłaby więc u Czytelnika pożądana. Nie jest jednak wymagana ani specjalna biegłość, ani obycie z dialektem ZX Spectrum. Jeśli książka ta stanie się dla Czytelnika inspiracją do własnych eksperymentów i zachętą do głębszego poznania ZX Spectrum, autor uzna swój cel za osiągnięty.

ZX Spectrum i reszta świata

Od początku istnienia ZX Spectrum produkowano kilka jego wersji. Z punktu widzenia użytkownika różniły się one tylko pojemnością pamięci RAM (zapis — odczyt) oraz obudową. Początkowo wytwarzano ZX Spectrum z pamięcią RAM o pojemności 16 KB (kilobajtów, 1 KB=2^{10}=1Ø24 bajty). Później produkowano równolegle wersje 16 KB i 48 KB. Wszystkie warianty z pamięcią 16 KB można rozbudować do pojemności 48 KB. Najbardziej kłopotliwe jest to w wersji pierwszej. Potrzebna jest dodatkowa płytka. W wersjach późniejszych uwzględniono możliwość przyszłej rozbudowy. Na płytce drukowanej znajdują się przygotowane podstawki. Wystarczy wetknąć w nie 12 dodatkowych układów scalonych i ewentualnie przelutować zworki na płytce. Liczne firmy oferują gotowe zestawy do samodzielnego rozszerzenia pamięci ZX Spectrum. Potrzebne układy scalone są zresztą łatwo dostępne na rynku. Można je także nabyć w wyspecjalizowanych sklepach w kraju.

Istnieje możliwość powiększenia pamięci RAM w Spectrum aż do 80 KB. Nie oznacza to jednak, że cała ta pamięć jest dostępna równocześnie. W praktyce 16 KB jest wspólne, reszta zaś pamięci dzieli się na dwa przełączane bloki po 32 KB. Ponieważ w koncepcji ZX Spectrum nie przewidziano przełączania bloków pamięci, jego praktyczne wykorzystanie wiąże się z pewnymi ograniczeniami i jest najbardziej opłacalne przy pracy z dużymi zbiorami danych. Przełączanie bloków odbywać się może ręcznie — przełącznikiem — lub programowo, przy nieco większych nakładach.

Największym mankamentem ZX Spectrum była jego niewygodna, nietypowa klawiatura. Pierwszymi próbami zaradzenia temu były dodatkowe klawiatury zewnętrzne. Najprostszą nasadzało się po prostu z góry na istniejącą klawiaturę gumową. Ergonomiczniej ukształtowane, większe klawisze plastikowe zapewniały palcom większą wygodę. Ponieważ jednak ich działanie sprowadzało się do naciskania położonych pod nimi klawiszy gumowych, pozostawał problem nie zawsze pewnego styku oraz ograniczonej trwałości. Lepsze są specjalne, dodatkowe klawiatury z własnymi elementami stykowymi. Część z nich dołączać można niezależnie od wbudowanej klawiatury, za pomocą złącza wkładanego w gniazdo z tyłu obudowy mikrokomputera.

Niektóre firmy oferują klawiatury wkomponowane w nową obudowę mikrokomputera. Przeróbka polega na wyjęciu płytki montażowej ze starej obudowy i zainstalowaniu jej w nowym pudełku z wygodną klawiaturą.
Firma Sinclair zareagowała dopiero w 1984 roku, wypuszczając na rynek ZX Spectrum+. Pod względem elektrycznym jest on identyczny ze starym ZX Spectrum 48 KB. Spectrum+ ma zupełnie nową obudowę i bardziej rozbudowaną klawiaturę z plastikowymi klawiszami. Pozostawiono jednak dotychczasową zasadę pracy klawiatury i dlatego, mimo że wygodniejsza od starej, nie jest ona w stanie dorównać typowej klawiaturze maszyny do pisania.
Podstawowym osprzętem do ZX Spectrum jest niewątpliwie magnetofon kasetowy. ZX Spectrum nie jest w tym względzie wybredny. Współpracuje dobrze nawet ze sprzętem niższej klasy. Ważna jest sprawna praca układu przesuwu taśmy. Wymagania co do właściwości toru elektronicznego są niewygórowane. Bardzo dobrze sprawdził się we współpracy ze Spectrum krajowy magnetofon B 113 Kapral. Nieźle pracuje też MK 232. Niektóre magnetofony wyższej klasy, zwłaszcza stereofoniczne, cechują się większą podatnością na zakłócenia i przekłamania informacji. Korzystając z magnetofonu stereofonicznego wskazane jest nagrywanie równocześnie na obydwie ścieżki i odtwarzanie także z wykorzystaniem obydwu. Zmniejsza to istotnie ryzyko przekłamań. Podstawowym czynnikiem, zapewniającym niezawodną współpracę z magnetofonem, jest jego staranna i systematyczna konserwacja. System przewijania i prowadzenia taśmy powinien być wyregulowany i nie może rozciągać taśmy. Zarówno głowica, jak i rolka dociskowa wraz z wałkiem powinny być często czyszczone — nie rzadziej niż raz na 20 godzin pracy. Ustawienia głowicy można z zadowalającym skutkiem dokonać na słuch, wyszukując ustawienia, przy którym dźwięk przesłuchiwanego programu jest najostrzejszy. Regulator barwy tonu należy ustawić w pozycji maksymalnego uwypuklenia sopranów i nastawić dużą głośność. Czułość słuchu na zmiany barwy dźwięku jest wtedy większa.
Jakość taśm używanych do zapisu informacji jest istotna. Parametry częstotliwościowe, niskoszumność, czułość i zniekształcenia nieliniowe nie są decydujące, ważne są natomiast: brak zaników (drop-out), stabilność czasowa oraz wytrzymałość mechaniczna. Z tego powodu odradza się taśmy cieńsze niż 18 μm (maksymalnie C-60). Dobre są kasety C-15. Oprócz skrócenia czasu przewijania szanują one bardziej taśmę niż kasety z taśmą większej długości. Kasety z nośnikiem żelazowym są przeważnie najlepsze i przewyższają często niezawodnością kasety chromowe. Bardzo dobrze spisują się kasety TDK i Maxell, z krajowych zaś — Super Ferro.
Kasety należy przechowywać w warunkach pyłoszczelnych i chronić przed gwałtownymi zmianami temperatury i polami magnetycznymi. Nie należy np. kłaść kaset na odbiorniku TV. Nawet słabe, zmienne pole magnetyczne, wytwarzane przez układy odchylania, może po pewnym czasie przyczynić się do utraty informacji. W żadnym razie nie wolno dopuścić do załamywania

taśmy ani dotykać powierzchni taśmy palcami. Zapis na taśmie z załamaniami najprawdopodobniej zostanie przekłamany, mimo że przy przesłuchaniu nagranej muzyki nie wykryje się zniekształceń. Przyczyną kłopotów z zapisem lub odczytem może być też niepewny kontakt w gnieździe wtykowym lub sfatygowany przewód połączeniowy magnetofonu.

Oprócz magnetofonu ważnym uzupełnieniem ZX Spectrum jest drukarka. Najtańsza jest oferowana przez firmę Sinclair drukarka ZX Printer. Wykorzystuje ona specjalny, metalizowany papier. Jakość wydruku nie jest najlepsza, papier jest drogi i wrażliwy. Sama drukarka nie cechuje się nadzwyczajną trwałością. W warunkach amatorskich, przy sporadycznych wydrukach, jest jednak rozwiązaniem wystarczającym. Często spotykana jest drukarka termiczna Alphacom 32. Jest ona nieznacznie droższa, lecz solidniejsza, pracuje też szybciej — do 2 wierszy na sekundę. Także i tu niezbędny jest specjalny, termoczuły papier. Wydruk ma barwę niebieskawą i nie jest zbyt kontrastowy. Podłoże jest jednak względnie odporne na dotyk palców itp., cena papieru jest poza tym niższa niż dla ZX Printer.

Trochę kosztowniejszym, lecz znacznie sprawniejszym urządzeniem jest drukarka Seikosha GP50S. Szerokość papieru ograniczona jest wprawdzie do 125 mm, jednak drukarka pracuje praktycznie na dowolnym, cienkim papierze. Prędkość wydruku wynosi ok. 1 wiersz/s. Jakość wydruku zależy głównie od stanu taśmy barwiącej. Może on być dzięki temu bardzo kontrastowy. Koszty eksploatacji GP50S są niższe niż obu drukarek omówionych poprzednio. Mankamentem GP50S jest nieznaczne rozciąganie wydruku w trybie graficznym — skala osi Y jest inna niż na ekranie.

Wspólną zaletą wszystkich drukarek omówionych powyżej jest możliwość ich bezpośredniego podłączenia do ZX Spectrum. Drukarki akceptują wszystkie rozkazy obsługi druku wbudowane w BASIC.

Tam, gdzie potrzebna jest drukarka pracująca na normalnym papierze formatu A4 lub na perforowanym papierze składankowym, zaczynają się kłopoty. Spectrum nie dysponuje żadnym ze standardowych interfejsów. Interfejs taki trzeba dopiero dołączyć z zewnątrz.

Firma Sinclair oferuje standardową przystawkę Interface 1, którą montuje się pod dnem obudowy komputera. Interface 1 zawiera standardowe złącze RS232, pozwalające dołączyć większość typowych drukarek. Często stosowane są np. drukarki mozaikowe Seikosha GP500AS oraz GP550AS. Podobne w konstrukcji, drukują na perforowanym papierze składankowym, GP550AS zaś pozwala także zakładać zwykły, nie perforowany papier w postaci luźnych kartek.

Interface 1 umożliwia także wymianę informacji między kilkoma, a nawet kilkudziesięcioma mikrokomputerami ZX Spectrum połączonymi w sieć. Oprócz tego niweluje istotny mankament, jakim jest brak szybkiej pamięci masowej. Interface 1 pozwala dołączyć do 8 jednostek tzw. mikrodrajwów (microdrive). Są to szybkie jednostki pamięci taśmowej, wykorzystujące kasety z wąską tasiemką magnetyczną w postaci pętli bez końca. Taśma

przewija się tylko w jedną stronę. Odbywa się to jednak bardzo szybko. Czas dostępu do zbioru danych nie przekracza kilku do kilkunastu sekund. Pojemność pojedynczej kasety wynosi ok. 95 KB. Pewną wadą mikrodrajwów jest względnie szybka zużywalność delikatnej taśmy, niewymienność z innymi nośnikami informacji i stosunkowo wysoka cena zapisanych na nie informacji, przekraczająca kilkakrotnie cenę zapisania tych samych danych na dyskietce.

Interface 1 zawiera dodatkowo 8 KB pamięci ROM, które włączane są przy obsłudze złącza RS232, sieci lub mikrodrajwów. W pamięci tej zapisane są procedury obsługi normalnie niedostępne w Spectrum, chociaż naniesione na klawiaturze rozkazów: FORMAT, CAT itp. Oprócz Interface 1 produkowane jest też tańsze rozszerzenie: Interface 2. Polecić można je jednak tylko amatorom gier: zawiera ono dwa gniazda dla manetek (joysticków) oraz gniazdo dla modułu typu ROM. W postaci modułów ROM dostępne są niektóre gry. Zamiast długiego ładowania gry z taśmy wystarczy włożyć moduł do odpowiedniego gniazda w Interface 2 i włączyć komputer. Niestety, moduły są drogie, a zawarte na nich gry raczej niższej klasy. W sumie więc możliwości Interface 2, zwłaszcza w poważniejszych zastosowaniach, są znikome.

Firma Sinclair postawiła zdecydowanie na mikrodrajwy. Tymczasem wielu producentów oferuje dla ZX Spectrum stacje dysków elastycznych, zarówno w formacie 5.25 jak również 3.5 cala. Najczęściej wraz ze stacją oferowany jest specjalny interfejs, zawierający wbudowaną pamięć ROM z oprogramowaniem obsługującym dyski. W niektórych rozwiązaniach rozkazy odwołujące się do stacji dysków są identyczne z tymi dla mikrodrajwów, w innych producent wprowadza nowe, specjalizowane instrukcje. Dołączenie stacji dysków elastycznych znacznie zwiększa przydatność Spectrum w zastosowaniach półprofesjonalnych. Dyskietka jest znacznie pewniejszym nośnikiem informacji niż kaseta mikrodrajwu, mieści też na ogół co najmniej dwukrotnie więcej informacji. Dyski elastyczne są szczególnie przydatne przy zastosowaniu mikrokomputera do obsługi banków danych.

Istnieje wiele przystawek rozszerzających możliwości Spectrum: pulpity graficzne, programatory pamięci EPROM, przetworniki pomiarowe i interfejsy sterujące urządzeniami zewnętrznymi: modelem robota, kolejką elektryczną, domowym sprzętem RTV. Wiele z tych przystawek może niewielkim kosztem wykonać średnio zaawansowany elektronik-amator. Przykładem może być pióro świetlne, składające się w najprostszym przypadku z kilkunastu elementów elektronicznych. ZX Spectrum nie zawiera sprzętowych układów obsługi pióra świetlnego. Urządzenie to w przypadku Spectrum nie cechuje się więc najwyższą sprawnością. Znacznie bardziej godny uwagi może być natomiast dodatkowy, programowany i wielogłosowy generator akustyczny, wzbogacający skromne możliwości muzyczne mikrokomputera. Generatory dostępne są często w zestawach do samodzielnego montażu i zawierają czasem dodatkowy interfejs.

Łatwy do samodzielnego wykonania jest też interfejs pozwalający na współpracę Spectrum z szeregową magistralą IEC. Magistralę taką wykorzystują standardowe urządzenia peryferyjne systemu VIC20/C-64/C-128. Wybór tych urządzeń jest bogaty, a ich ceny często znacznie niższe od analogicznej klasy sprzętu przeznaczonego dla Spectrum. Przykładem może być plotter-printer VC1520. Urządzenie to, zbliżone cenowo do GP50S i pracujące na papierze podobnej szerokości, kreśli rysunki czterobarwnymi liniami ciągłymi lub przerywanymi o nastawialnej gęstości. Może także pisać teksty, mieszcząc 10, 20, 40 lub 80 znaków w wierszu, poziomo lub pionowo. Zarówno zmiany koloru pisaka, jak i wielkości, orientacji liter, trybu pracy odbywają się programowo. Jednym słowem, w warunkach amatorskich jest VC1520 sprzętem bardzo uniwersalnym i "inteligentnym" (zawiera wbudowany mikrokomputer). Innym, przydatnym elementem systemu Commodore może być stacja dysków elastycznych 5 1/4 cala typu VC1541. Mimo że stosunkowo wolna, mieści ok. 170 KB na jednej stronie dyskietki i jest najtańszą z dostępnych na rynku stacji dysków do komputerów domowych. Także i ona zawiera wbudowany mikrokomputer, realizujący samodzielnie liczne, złożone zadania związane z obsługą informacji na dysku, odciążając główny mikrokomputer, z którym współpracuje.

Mikrokomputer ZX Spectrum jest urządzeniem względnie niezawodnym, pod warunkiem pewnej kultury obsługi. Stanowczo odradza się, częstego przy grach, siłowego walenia w klawiaturę. Należy unikać nakrywania pracującego Spectrum — łatwo wtedy o przegrzanie. Szczelinę złącza krawędziowego z tyłu obudowy trzeba chronić przed kontaktem z metalowymi przedmiotami. Wałęsający się beztrosko wtyk przewodu słuchawkowego stał się już nieraz przyczyną uszkodzenia mikrokomputera, wywołując przypadkowe zwarcie. Przy dołączaniu do wspomnianego złącza różnych przystawek trzeba pamiętać o obowiązkowym wyłączeniu zasilania. Złącze to nie jest zbyt solidne i przy drganiach układu komputer — przystawka może nie zapewniać pewnego kontaktu. Jest to przyczyną nie zawsze wytłumaczalnych wypadnięć systemu. Warto więc czasem połączenie mikrokomputera z przystawką usztywnić.

Momentem w eksploatacji krytycznym, sprzyjającym powstawaniu uszkodzeń, jest wkładanie wtyku zasilania. Należy robić to energicznie, nie dopuszczając do iskrzenia. Podobnie z wyłączaniem. ZX Spectrum nie ma niestety przycisku RESET, przywracającego bez wyłączania zasilania wyjściowy stan systemu. Wyłączenie i ponowne włączenie zasilania jest więc czasem jedynym wyjściem. Jeśli jesteśmy na poziomie języka BASIC, efekt analogiczny jak przy wyłączeniu zasilania, jednak bez jego ujemnych następstw, osiągnąć można zleceniem **RAND USR Ø**. Wersję ZX Spectrum+ wyposażono już we wbudowany przycisk RESET.

Dobrymi zastosowaniami dla ZX Spectrum mogą być podręczne obliczenia matematyczne. Możliwości graficzne pozwalają na przejrzyste i efektowne przedstawianie wyników w postaci graficznej. Spectrum jest doskonałym

komputerem do nauki języków programowania. Dostępne są do niego m.in. takie języki, jak: ASEMBLER, PASCAL, FORTH, LOGO, MikroPROLOG, C, a nawet LISP.

Także w zastosowaniach edukacyjnych i samokształceniu ZX Spectrum może wyświadczyć duże usługi. Paleta dostępnego oprogramowania jest tu bardzo szeroka. W pracowni hobbisty jest ZX Spectrum również doskonałym narzędziem. Względna prostota systemu pozwala na jego opanowanie i nagięcie do własnych wymagań.

Odradzać można natomiast użycie ZX Spectrum do zastosowań związanych z obróbką tekstów, np. redagowaniem artykułów lub korespondencji. Nawet po zamontowaniu wygodniejszej klawiatury przeszkodą będzie niewielka pojemność ekranu w trybie tekstowym. Przy użyciu specjalnego oprogramowania można uzyskać co prawda 64 znaki w wierszu, czytelność tych znaków jest jednak problematyczna i praca przy ekranie staje się męcząca.

W dziedzinie rozrywki jest Spectrum komputerem doskonałym. Liczne gry komputerowe reprezentują wysoką klasę i stały sę wzorcem dla podobnych programów oferowanych dla innych systemów. A partyjki szachowej z komputerem nie odmówi sobie niekiedy nawet najbardziej zatwardziały profesjonalista!

Repetytorium z języka BASIC w ZX Spectrum

Język BASIC w ZX Spectrum ma wiele nietypowych elementów i właściwości, odbiegając od innych, rozpowszechnionych dialektów, np. MICROSOFT--BASIC. Znajomość specyfiki języka ułatwi nam samodzielne układanie efektywnych programów. Będzie niezbędna także i przy adaptacji dla Spectrum programów z innych maszyn i podczas przenoszenia opracowanych dla Spectrum programów na inne komputery. Przypomnijmy zatem słownik Spectrum, zwracając uwagę zarówno na jego słabości, jak i na atuty.

Budowa programu

Program składa się z numerowanych wierszy, zawierających instrukcje. Numer wiersza musi być liczbą całkowitą z zakresu 1—9999. Każda instrukcja musi rozpoczynać się rozkazem (nazwą instrukcji), czyli słowem określającym rodzaj czynności. Spectrum akceptuje także instrukcje wprowadzane bez numeru wiersza. Takie instrukcje, wprowadzane i natychmiast wykonywane w tzw. trybie bezpośrednim, nazywamy zleceniami.

Zarówno rozkazy, jak i inne słowa kluczowe (zastrzeżone słowa, o ściśle określonym w języku BASIC znaczeniu) wprowadzamy pojedynczym naciśnięciem klawisza. Dotyczy to m.in. nazw funkcji standardowych. Częstym błędem u początkujących jest wypisywanie nazwy funkcji jako ciągu liter. To samo dotyczy operatorów (symboli operacji): <>, >=, <=. Je także wprowadzamy pojedynczym klawiszem, uprzednio nacisnąwszy SYMBOL SHIFT.

W jednej linii programu umieścić można wiele instrukcji rozdzielonych dwukropkami. Spacje (odstępy) umieszczone zarówno wewnątrz instrukcji, jak i bezpośrednio po numerze wiersza, są w programie zachowywane. Ułatwia to uwypuklanie struktury programu tzw. wcięciami. Pierwsza spacja po numerze wiersza jest utajniana. Widać ją tylko wtedy, gdy na jej miejscu pojawi się kursor. Aby więc przesunąć instrukcję w prawo np. o dwa pola, trzeba między numerem wiersza a rozkazem wcisnąć spację trzy razy.

Nazwy zmiennych

ZX Spectrum operuje dwoma typami zmiennych: liczbowymi (numerycznymi) i tekstowymi (łańcuchowymi, stringowymi).
Wszystkie zmienne liczbowe są typu rzeczywistego. Dopuszczalny zakres wartości wynosi od 4×10^{-39} do 10^{38}. Liczby mniejsze niż 4×10^{-39} są reprezentowane przez 0.
Nazwy zmiennych liczbowych prostych mogą mieć dowolną długość. Wszystkie znaki różne od spacji są istotne. Spacje zawarte wewnątrz nazwy zmiennej są ignorowane. Nazwy: **WYNIK TESTU, WYNIK TESTU** i **WYNIKTESTU** oznaczają więc tę samą zmienną. Spectrum nie rozróżnia w nazwach zmiennych małych i dużych liter. Nazwy: **ALFA1, alfa1** i **Alfa1** symbolizują więc tę samą zmienną liczbową. Podobnie **i$** i **I$** oznaczają tę samą zmienną tekstową.
Zmienne sterujące w pętlach **FOR... NEXT**, mimo że także należą do kategorii zmiennych numerycznych prostych, muszą mieć niestety nazwy jednoliterowe. To samo dotyczy nazw zmiennych tekstowych oraz wszystkich tablic.

Instrukcje ogólnego zastosowania

LET

```
LET X=5+(alfa-X)/7↑2-SQR X
LET Y=6    : LET X=X+1
```

Instrukcja przypisania, zwiastowana rozkazem **LET**, nadaje zmiennej wskazanej po lewej stronie znaku = (symbolu przypisania, podstawienia) wartość uzyskaną z wyliczenia wyrażenia umieszczonego po prawej stronie. Pojedyncza stała lub zmienna także uważana jest za wyrażenie.
Dopuszczalne w innych dialektach opuszczenie rozkazu **LET** nie jest w Spectrum dozwolone.
Wyrażenie arytmetyczne składa się zazwyczaj ze stałych i zmiennych liczbowych oraz funkcji rozdzielonych operatorami (symbolami operacji) dodawania +, odejmowania −, mnożenia *, dzielenia / i potęgowania ↑. Priorytety operatorów — mówiąc prościej, kolejność wykonywania działań — są takie, jak przyjęto w matematyce. Najpierw wykona się potęgowanie, następnie mnożenie i dzielenie, na końcu dodawanie i odejmowanie. Jeśli w wyrażeniu występuje kilka operatorów o równych priorytetach, operacje wykonane będą w kolejności od lewej do prawej. Tak więc przy obliczaniu wartości wyrażenia:

A*5/D*T

najpierw wyznaczony będzie iloczyn wartości zmiennej **A** i stałej **5**, potem otrzymany iloczyn zostanie podzielony przez wartość zmiennej **D**, a dopiero na końcu całość pomnożona przez wartość zmiennej **T**.

Używając nawiasów okrągłych ( ) można zmienić kolejność wykonywania działań. Jako pierwsze wykonają się operacje zawarte w najbardziej wewnętrznej (najgłębiej zagnieżdżonej) parze nawiasów.
Stosując potęgowanie trzeba wiedzieć, że niedopuszczalne jest podnoszenie do potęgi liczb ujemnych. Nawet w tak powszechnie występującym wyrażeniu, jak x^2 (w języku BASIC: **x↑2**), ujemna wartość zmiennej **x** doprowadzi do błędu i przerwania wykonywania programu. W trakcie potęgowania komputer musi wykonać wiele mnożeń i dodawań. Zastępując potęgowanie mnożeniem (**A*A** zamiast **A↑2**), możemy znacznie przyspieszyć wykonywanie programu.

Funkcje arytmetyczne:

ZX Spectrum dysponuje bogatym repertuarem funkcji arytmetycznych:

ABS — wartość bezwzględna z argumentu,
ACS — arcus cosinus argumentu (wynik w mierze łukowej),
ASN — arcus sinus argumentu (wynik jw.),
ATN — arcus tangens argumentu (wynik jw.),
COS — cosinus argumentu wyrażonego w mierze łukowej,
EXP — funkcja wykładnicza e^x,
INT — część całkowita argumentu. Wartością funkcji jest największa liczba całkowita nie przewyższająca wartością argumentu. Przy argumencie dodatnim INT "odcina" część ułamkową,
LN — logarytm naturalny argumentu. Aby uzyskać logarytm dziesiętny argumentu, wystarczy wynik podzielić przez **LN 10**,
RND — funkcja losowa (właściwie: pseudolosowa). Jej wartością jest liczba rzeczywista z przedziału 0—1 (0—0.99999999),
SGN — funkcja znaku. Przyjmuje tylko trzy możliwe wartości: −1 dla argumentów ujemnych, 0 dla argumentu zerowego i +1 dla dodatniego,
SIN — sinus argumentu wyrażonego w mierze łukowej,
SQR — pierwiastek kwadratowy z nieujemnego argumentu,
TAN — tangens argumentu wyrażonego w mierze łukowej (radianach),
PI — funkcja o wartości stałej, równej 3,1415927.

Funkcje **RND** i **PI** nie wymagają żadnych argumentów. W przypadku pozostałych funkcji argument nie musi być ujęty w parę nawiasów, jak w innych dialektach. Zamiast:

```
LET Y=SQR(X)-SIN(Y)
```

można zapisać:

```
LET Y=SQR X-SIN Y
```

Trzeba jednak pamiętać, że za argument uznany zostanie tylko pierwszy

obiekt umieszczony na prawo od nazwy funkcji. Przy wykonywaniu instrukcji:

```
LET alfa=SIN X+1
```

najpierw wyliczona zostanie wartość funkcji sinus z **X**, a dopiero do wartości funkcji dodana zostanie jedynka. Niekiedy zamknięcie argumentu w nawiasach jest więc niezbędne, np.:

```
LET X1=SIN(Y1/(SIN(X-1)))
```

Umieszczanie argumentów funkcji w nawiasach, nawet gdy niekonieczne, polepsza znacznie przejrzystość programu, jest więc praktyką godną polecenia. Język BASIC dopuszcza użycie wyłącznie nawiasów okrągłych.
Jeśli w wyrażeniu występują funkcje, najpierw wyliczane są argumenty funkcji, następnie wartości funkcji. Dopiero potem wykonywane są operacje arytmetyczne.

Wyrażenia arytmetyczno-logiczne

Oprócz wyrażeń arytmetycznych mogą w Spectrum występować wyrażenia logiczne oraz mieszane, arytmetyczno-logiczne.
Wyrażenie logiczne może mieć jedynie dwie wartości: "prawda" (ang. true) lub "fałsz", "nieprawda" (ang. false). Wyrażenia logiczne występują w programach z reguły jako równości lub nierówności i składają się z dwóch wyrażeń arytmetycznych, rozdzielonych operatorem logicznym (relacji, operatorem porównania). Dopuszczalne są następujące operatory porównania: =, <, >, <>, >=, <=. Przykład:

```
1+A=10/2
```

Wartością logiczną tego wyrażenia będzie "prawda" tylko wtedy, gdy zależność zostanie spełniona, tzn. w przypadku, gdy wartością **A** będzie 4. W razie niespełnienia warunku wartością logiczną jest "fałsz".
Postawiony warunek może być tylko spełniony lub nie spełniony — stąd jedynie dwie możliwe wartości logiczne.
Choć w matematyce logiczna "prawda" i "fałsz" nie mają interpretacji arytmetycznej, w Spectrum wartości logiczne przedstawiane są za pomocą odpowiednich wartości liczbowych. "Fałsz" jest reprezentowany przez Ø, "prawda" przez 1. W praktyce, przy próbie interpretacji liczby jako wartości wyrażenia logicznego, każda niezerowa wartość uznana będzie za "prawdę".
Pozornie bezsensowna instrukcja:

```
LET R=X-5=V
```

jest w Spectrum najzupełniej poprawna. Znaczy ona po prostu, że zmiennej **R** przypisana będzie wartość arytmetyczna (liczba), odpowiadająca logicznej ocenie warunku:

```
X-5=V
```

Przy spełnieniu zależności, czyli we wszystkich przypadkach, gdy wartość **X** będzie o 5 większa od **V**, zmienna **R** otrzyma wartość 1. W przeciwnym razie zmiennej **R** zostanie przypisana wartość Ø. Priorytet operatorów porównania jest niższy od operatorów arytmetycznych, zatem operacje logiczne wykonują się dopiero po wykonaniu wszystkich działań arytmetycznych.

Jeszcze niższy priorytet mają operatory logiczne **NOT**, **AND**, **OR**. Wykonywane są w takiej kolejności, w jakiej je wymieniono. Operator negacji logicznej **NOT** zmienia na przeciwną wartość logiczną wyrażenia, przed którym został umieszczony. Zamiast "prawdy" otrzymamy "fałsz" i na odwrót:

```
LET X=NOT 4/2=2
```

Warunek spełniony jest zawsze. **NOT** neguje wartość logiczną, zmienna **X** otrzyma więc wartość Ø, reprezentującą "fałsz". Jak widać, możliwe jest wykorzystanie **NOT** w wyrażeniu arytmetycznym:

```
LET Z=NOT X
```

Wartość zmiennej **X** będzie przez operator **NOT** traktowana jako wartość logiczna. Zmienna **Z** otrzyma więc wartość 1 dla **X**=Ø i wartość Ø dla pozostałych wartości **X**. Powyższą instrukcję można oczywiście zapisać przejrzyściej:

```
LET Z= X=Ø
```

AND jest operatorem iloczynu logicznego. Wartość logiczna iloczynu dwóch lub więcej wyrażeń logicznych będzie prawdą tylko wtedy, gdy wszystkie składowe wyrażenia logicznego też będą prawdziwe:

```
LET Z= A=5 AND X>=Ø AND X<=1Ø
```

Wartością **Z** będzie 1 tylko wówczas, gdy spełnione zostaną równocześnie wszystkie trzy warunki: wartość **X** zawarta będzie w przedziale ⟨Ø, 1Ø⟩, a wartością **A** będzie 5.

OR symbolizuje sumę logiczną. Wartość logiczna sumy logicznej dwóch lub więcej wyrażeń logicznych będzie prawdą wtedy, gdy prawdziwe będzie przynajmniej jedno ze składowych wyrażeń logicznych:

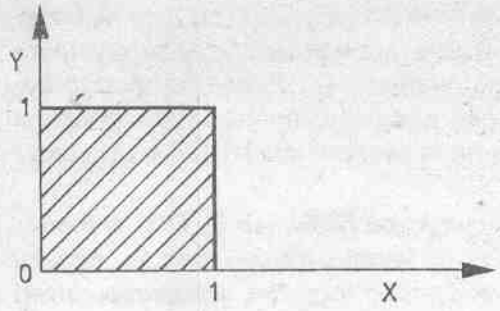
```
LET Z= A=1 OR A=2 OR A=3
```

Wartość **Z** stanie się jedynką, gdy wartością **A** będzie jedna z liczb całkowitych: 1, 2 lub 3.

Budując bardziej złożone wyrażenia logiczne posłużyć się można nawiasami okrągłymi:

```
LET W= (X<=1 OR X>=Ø) AND (Y<=1 OR Y>=Ø)
```

Wartością **W** będzie 1 tylko wtedy, gdy punkt o współrzędnych **X**, **Y** należeć będzie do kwadratu z rys. 1.



Rys. 1 Punkty spełniające powyższy warunek

Wyrażenia łańcuchowe

Instrukcja przypisania może być użyta do nadawania wartości zmiennym tekstowym:

```
LET A$="ABCDE" : LET B$=A$+V$
```

Po lewej stronie znaku = musi wystąpić nazwa zmiennej tekstowej, po prawej — dowolne wyrażenie tekstowe (łańcuchowe, stringowe). Jedynym operatorem w wyrażeniach tekstowych może być operator "sklejania" (złożenia, konkatenacji) łańcuchów, symbolizowany przez +. Innymi operacjami na łańcuchach zajmiemy się później.
Wszystkie operatory relacji (porównania) stosuje się także do tekstów:

```
LET A= A$="Beta"+"Basic"
```

Zmienna liczbowa **A** będzie 1 przy spełnieniu warunku, czyli gdy treścią zmiennej tekstowej **A$** będzie "BetaBasic". W odróżnieniu od interpretacji nazw zmiennych, przy porównaniu tekstów małe i duże litery są rozróżniane. "Alfa" i "alfa" są więc różnymi (nieidentycznymi) tekstami.
Znaczenie operatorów: = i <> jest w odniesieniu do tekstów oczywiste: "identyczny" i "nieidentyczny". Dwa teksty są identyczne wtedy, gdy są równej długości (uwzględniając także spacje) i wszystkie znaki na odpowiednich pozycjach pokrywają się.
Operatory: < i > badają alfabetyczne uporządkowanie łańcuchów. Warunek:

```
A$ < B$
```

jest spełniony wtedy, gdy tekst **A$** poprzedza w sensie alfabetycznym tekst **B$** (np. A$="ALFA", B$="GAMMA"). Porównywanie tekstów polega na porównywaniu kolejnych par znaków w obu tekstach. Jeżeli pierwsze znaki są identyczne, porównuje się następne, aż do wykrycia różnicy lub osiągnięcia końca jednego lub obu tekstów. W razie wykrycia niezgodności znaków za poprzedzający alfabetycznie uznany będzie łańcuch, w którym wystąpił znak o mniejszym kodzie. Tkwi tu pułapka: małe litery mają kody większe od liter dużych. Warunek:

"ala" < "Ola"

na pozór jest spełniony. Litera "a" występuje w alfabecie przed "O". Kod "a" jest najmniejszy spośród kodów małych liter. Duże "O" ma jednak kod mniejszy od "a", dlatego ZX Spectrum uzna warunek za nie spełniony.
Wyrażenia arytmetyczne i tekstowe mogą wspólnie tworzyć wyrażenia logiczne:

```
LET R=A$="W" AND X>0
```

R będzie równe 1, gdy tekst **A$** będzie pojedynczą literą "W", a liczba **X** będzie dodatnia.
Możliwość arytmetycznej interpretacji wartości logicznej jest silnym narzędziem programistycznym. Zamodelujmy prędkość obrotową kół samochodu Polski Fiat 126 w zależności od obrotów wału korbowego przy braku poślizgu na sprzęgle. Niech **N** przedstawia obroty wału, **K** — prędkość obrotową tylnych kół. Sprawa byłaby prosta, gdyby nie skrzynia biegów. Niech **B** przyjmuje wartości od 1 do 4, symbolizujące włączony aktualnie bieg:

```
LET K=((B=1)*0.063+(B=2)*0.1+(B=3)*0.157+(B=4)*0.236) *N
```

Współczynniki 0.063, 0.1, 0.157 i 0.236 przedstawiają przełożenie dla biegów 1, 2, 3 i 4. Każdy z nich poprzedzony jest "włącznikiem" w postaci warunku logicznego. Spośród czterech warunków w danej chwili spełniony jest tylko jeden i tylko odpowiadający mu "włącznik" będzie miał arytmetyczną wartość 1 (pozostałe — 0). Zawartość nawiasu zachowuje się więc tak, jakby w danej chwili zawierał on tylko jeden współczynnik, odpowiadający aktualnemu biegowi. Pozostałe współczynniki, mnożone przez 0, nie mają wpływu na wynik obliczeń.

STOP **STOP**

Przerywa wykonywanie programu, wyświetlając informację o numerze wiersza i numerze instrukcji w wierszu, w której nastąpiło zatrzymanie. Wznowienie pracy programu możliwe jest zleceniem **CONTINUE**.

GO TO

```
GO TO 30
GO TO petla
GO TO 20*I+100
```

Skok bezwarunkowy **GO TO** powoduje wykonanie jako następnej pierwszej instrukcji z wiersza o numerze określonym wartością argumentu. W odróżnieniu od innych dialektów, w Spectrum argumentem **GO TO** może być dowolne wyrażenie o wartości liczbowej. Pozwala to np. stosować etykiety symboliczne. Niech w linii **500** zaczyna się procedura wyświetlania objaśniającego tekstu. Zamiast suchego **GO TO 500** możemy na początku programu zadeklarować **LET Objasnienie=500**, tam zaś, gdzie trzeba przejść do wyświetlania objaśnień, napisać: **GO TO Objasnienie**.

Gdy wartość argumentu **GO TO** będzie ułamkowa, zostanie zaokrąglona do najbliższej liczby całkowitej zgodnie z ogólnymi regułami. **3Ø.1** oraz **3Ø.49** zostaną zaokrąglone do **3Ø**, **1ØØ.5** i **1ØØ.999** do **1Ø1**. Wygodną, lecz niebezpieczną własnością **GO TO** w Spectrum jest fakt, że nie ma sygnalizacji nieznalezienia w programie linii o wskazanym przez argument numerze. Wykonywany jest w takim przypadku skok do linii o najbliższym, większym numerze.

GO SUB **GO SUB 1ØØØ : GO SUB wykres**

Wywołanie podprogramu **GO SUB** działa identycznie jak **GO TO**, a oprócz tego zapamiętuje miejsce w programie, w którym zostało wykonane. Ściślej, zapamiętuje położenie następnej instrukcji. Pozwala to na powrót z podprogramu do miejsca bezpośrednio następującego za wywołaniem. Podprogramem nazywamy ciąg instrukcji, tworzących funkcjonalną całość i wywoływanych przez **GO SUB** dla wykonania jakiejś typowej lub przynajmniej wyraźnie wyodrębnionej czynności w programie. Instrukcja **GO SUB** jest w istocie poleceniem wykonania czynności złożonej, o specyfice określonej treścią podprogramu.

RETURN **RETURN**

Instrukcja powrotu z podprogramu **RETURN** jest nieodzownym dopełnieniem **GO SUB**. Może wystąpić ona jedynie w podprogramie. **RETURN** pobiera ostatnio zapamiętane przez **GO SUB** dane o wywołaniu i powoduje skok do wskazanego miejsca. Dane o wywołaniu są usuwane, jako już niepotrzebne.
Podprogramy umieszczone mogą być w dowolnym miejscu programu. Wykonanie zawartych w nich instrukcji dozwolone jest jednak wyłącznie w wyniku wywołania **GO SUB**. Powrót z podprogramu powinien wykonywać się wyłącznie przez **RETURN**. Instrukcji **RETURN** w podprogramie może być więcej niż jedna, np. na zakończenie każdego z kilku wariantów pracy podprogramu. Sensowne jest nadawanie numerom wierszy, rozpoczynającym podprogramy, symbolicznych nazw. Przykład:

Program 1

```
  10 LET Xkwadrat=1000
  ..............
  70 LET X=2:    GO SUB Xkwadrat
  80 LET X=X+2: GO SUB Xkwadrat
  90 STOP
1000 LET X=X*X: RETURN
```

Przedstawiony program oblicza wartość wyrażenia $(x^2+2)^2$. Zastosowany podprogram podnosi do kwadratu aktualną wartość zmiennej **X**. Jeden podprogram można wywoływać z wnętrza drugiego. Mówimy wtedy o za-

gnieżdżaniu podprogramów. W pewnych warunkach podprogram może wywoływać nawet... sam siebie. Jest to tzw. rekursja. Głębokość zagnieżdżenia, czyli liczba kolejno wykonanych **GO SUB**, nie zakończonych jeszcze przez **RETURN**, jest w Spectrum ograniczona tylko pojemnością pamięci i może sięgać wielu tysięcy.

IF... THEN `IF X=Ø THEN GO TO 2Ø`
`IF X=1 AND A$="T" THEN LET Y=3`

Instrukcja warunkowa **IF... THEN** (jeżeli... to) bada wartość logiczną wyrażenia zawartego między **IF** a **THEN**. Gdy wyrażenie jest prawdziwe, wykonywana jest instrukcja (lub ich ciąg, aż do końca wiersza) umieszczona po **THEN**. W razie fałszu wszystkie instrukcje między **THEN** a końcem wiersza są ignorowane i program przechodzi do pierwszej instrukcji w wierszu następnym. Pamiętając o arytmetycznej interpretacji wyrażeń logicznych (i na odwrót), zapis:

IF X<>Ø THEN GO SUB 3ØØ

można, nie zmieniając jego sensu, przedstawić tak:

IF X THEN GO SUB 3ØØ

Obydwa wyrażenia przyjmą wartość Ø (fałsz) tylko dla **X**=Ø. Przy logicznej interpretacji wartości nie bada się, czy jest ona rezultatem czystych operacji logicznych, czy też mniej lub bardziej przewrotnych sztuczek. Dobry styl programowania zaleca unikanie sztuczek i zapis programu w przejrzystej, logicznej formie. Użycie owych sztuczek pozwala jednak niekiedy zoptymalizować krytyczny fragment, skracając czas wykonania.
Po słowie **THEN** musi następować rozkaz. Inne dialekty zezwalają na umieszczenie po **THEN** bezpośrednio numeru wiersza, do którego nastąpić ma skok. W ZX Spectrum zamiast np.:

IF X=2 THEN 4ØØ

zapisać trzeba:

IF X=2 THEN GO TO 4ØØ

FOR... TO... STEP `FOR i =1 TO 1Ø STEP - Ø2`
NEXT `FOR K=-5 TO 5 : LET Y=Y+K : NEXT K`

Instrukcje **FOR...** i **NEXT** służą do organizacji pętli programowych (cykli), działając na podobieństwo pary nawiasów. Wszystkie zawarte pomiędzy nimi instrukcje zostaną powtórzone zadaną liczbą razy, która określona jest pośrednio w nagłówku pętli, tworzonym przez **FOR... TO... STEP**. Wskazana za **FOR** zmienna liczbowa prosta jest zmienną sterującą pętli. Musi ona mieć nazwę jednoliterową. Przy rozpoczynaniu pętli zmiennej tej przypisywana jest wskazana wartość. Za każdym powtórzeniem pętli wartość

zmiennej zwiększana jest o zadany przyrost (krok), określony po słowie **STEP**. Pętla jest powtarzana, aż zmienna sterująca nie osiągnie wartości granicznej, zadanej wyrażeniem po słowie **TO**.
Najczęściej stosowany jest krok równy 1, zmienna sterująca przyjmuje wartości kolejnych liczb całkowitych. Gdy krok=1, można opuścić człon **STEP1**. Poniższa pętla oblicza np. wartość sumy kwadratów wszystkich liczb całkowitych od 1 do 75:

Program 2

```
10 LET S=0
20 FOR X=1 TO 75
30   LET S=S^2
40 NEXT X
```

Celowe jest wcinanie pętli. Polega ono na cofnięciu w prawo wszystkich linii programu zawartych między **FOR**... a **NEXT**. Kosztem nieznacznego zwiększenia zajętości pamięci zabieg ten podnosi przejrzystość programu, pozwalając na pierwszy rzut oka określić zakres działania pętli.
W instrukcji **NEXT** badanie warunku zakończenia pętli następuje dopiero po zwiększeniu wartości zmiennej sterującej. Wskutek tego zmienna ta po zakończeniu pętli ma wartość o krok większą od ostatniej wartości, dla której pętla się wykonała. Np. w programie 2 wartość zmiennej **X** po zakończeniu pętli wynosi 76. Ważną własnością **FOR**... **NEXT** w Spectrum jest badanie warunku zakończenia pętli już przed pierwszym jej wykonaniem. Gdy warunek zakończenia jest spełniony już na wstępie, instrukcje zawarte między **FOR**... a **NEXT** nie wykonają się ani razu:

Program 3

```
10 LET R=10
20 FOR I=2 TO 1
30   LET R=R*R
40 NEXT X
```

Krok domyślnie wynosi 1. Przy dodatnim kroku wartość zmiennej sterującej już na wstępie przewyższa wartość graniczną. Pętla nie zostanie wykonana, zmienna **R** zachowa wartość 10. Większość innych dialektów zachowuje się odmiennie. Pętla wykona się zawsze przynajmniej raz.
Badanie warunku zakończenia odbywa się tam wyłącznie instrukcją **NEXT**. Wyrażenia określające krok i wartość graniczną wyliczane są tylko raz — przy rozpoczęciu pętli. Nie można zmieniać warunku zakończenia już po rozpoczęciu pętli:

Program 4

```
10 LET X=10: LET S=0.2
20 FOR I=1 TO X STEP S
30   LET X=1: LET S=1
40 NEXT I
```

Mimo zmiany wartości **X** i **S** pętla wykonywać się będzie aż do **I=1Ø** z krokiem Ø.2. Możliwa jest natomiast zmiana wartości zmiennej sterującej wewnątrz pętli instrukcją przypisania. Dodanie instrukcji:

35 LET I=1Ø

kończy pętlę w programie 4 po jednym wykonaniu.
Podanie nazwy zmiennej sterującej po **NEXT** jest w Spectrum obowiązkowe. Nie wolno zamykać jedną instrukcją **NEXT** kilku pętli zagnieżdżonych. Zamiast:

NEXT X,Y,Z

trzeba zapisać:

NEXT X : NEXT Y : NEXT Z

Z wnętrza pętli **FOR... NEXT** można w Spectrum wyskoczyć instrukcją **GO TO** bez żadnych niekorzystnych konsekwencji. (Parametry pętli nie są przechowywane na stosie, lecz w odpowiednich polach zmiennej sterującej). Nie jest to dozwolone w innych komputerach. Skok z wnętrza pętli jest nieelegancki, czasem jednak upraszcza i przyspiesza program.

DATA	**DATA 1, −Ø.5 , "alfa"**
READ	**READ x, beta, A$**

Instrukcja **DATA** pozwala umieścić w programie dane o charakterze liczbowym i tekstowym. W jednej instrukcji **DATA** można zawrzeć wiele danych obu typów. Poszczególne elementy listy danych trzeba rozdzielić przecinkami. Dane tekstowe (stałe łańcuchowe) muszą być ujęte w znaki cudzysłowu. Wszystkie instrukcje **DATA** w programie można rozpatrywać jako jeden ciąg elementów ułożonych kolejno (sekwencyjnie). Dane ze zbioru **DATA** odczytywane są instrukcją **READ**. Po rozkazie **READ** wymienione są nazwy zmiennych, którym ma zostać przypisana wartość. Nazwy rozdzielone są przecinkami. W trakcie wykonywania instrukcji **READ** dane są odczytywane w takiej kolejności, w jakiej są ułożone. Poniższy program oblicza sumę dziesięciu liczb, zapisanych w instrukcjach **DATA**:

Program 5

```
10 LET S=0
20 FOR I=1 TO 10
```

```
30   READ X: LET S=S+X
40 NEXT I
50 DATA 1,2.3,14,-5,22
60 DATA 1E3,-2,3,4.5,6
```

Zmienna **I** jest tylko licznikiem, odliczającym 10 powtórzeń pętli. Instrukcja **READ** odczytuje po jednym elemencie z listy **DATA** i przypisuje go zmiennej **X**. Wartość ta następnie dodawana jest do sumy **S**, wyzerowanej wstępnie przed rozpoczęciem pętli.
Instrukcje **DATA** mogą być umieszczone w dowolnym miejscu programu, a nawet przeplatać się z innymi instrukcjami. Kolejność i typ danych na liście **DATA** muszą odpowiadać porządkowi ich odczytywania. Próba wykonania np. **READ a**, gdy kolejnym elementem listy **DATA** jest tekst, skończy się przerwaniem programu i komunikatem o błędzie.
Oryginalną własnością **DATA** i **READ** w ZX Spectrum jest możliwość umieszczania wśród danych oprócz stałych liczbowych i tekstowych (np. 1.5, "Ala") także wyrażeń arytmetycznych, logicznych i tekstowych:

Program 6

```
10 DATA 1,1/3,X+5,SQR (Z+3)-X
20 READ X, Y, Z, A
```

Zmienna **X** otrzyma wartość 1. Wartość wyrażenia **1/3** zostanie wyliczona i przypisana zmiennej **Y**. Zmiennej **Z** ma zostać przyporządkowana wartość wyrażenia **X+5**. Wartość tego wyrażenia zostanie wyliczona w chwili wykonywania instrukcji **READ** z uwzględnieniem aktualnych wartości wymienionych w wyrażeniu zmiennych. Zmienna **Z** otrzyma więc wartość 6. Wartość **A** także zostanie wyznaczona z wyrażenia (=2). Teraz jest już jasne, dlaczego w ZX Spectrum teksty na liście **DATA** muszą być ujęte w cudzysłowy. Tekst bez cudzysłowu zostałby uznany za nazwę zmiennej.

RESTORE **RESTORE**
RESTORE 200

Dane ze zbioru **DATA** mogą być potrzebne w programie wielokrotnie. Wykonanie instrukcji **RESTORE** powoduje odczytywanie danych od początku, od pierwszego elementu pierwszej linii **DATA** w programie. Niekiedy celowe może być odczytywanie tylko partii danych z pominięciem elementów początkowych. W tym przypadku można zastosować **RESTORE** z argumentem w postaci wyrażenia liczbowego. Argument ten wskazuje numer linii, od której zacznie się po najbliższej instrukcji **READ** poszukiwanie instrukcji **DATA**. Oto przykład: poniższy program określa, ile liczb w danym zbiorze różni się od średniej arytmetycznej w tym zbiorze nie więcej niż o dopuszczalną wartość. Wszystkie dane zebrane są w zbiorze **DATA**.

Pierwszą daną jest liczebność zbioru, drugą — dopuszczalne odchylenie od średniej:

Program 7

```
 10 READ N,D: LET S=0
 20 FOR I=1 TO N
 30   READ X: LET S=S+X
 40 NEXT I
 50 LET S=S/N
 60 RESTORE 110: LET L=0
 70 FOR I=1 TO N
 80   READ X: IF X>=S-D AND X<=S+D THEN LET L=L+1
 90 NEXT I
100 DATA 8, 0.75
110 DATA 13, 14.5, 9, 12, 11, 12.5, 11, 9.3
```

Najpierw należy wyznaczyć średnią. W tym celu wszystkie elementy zbioru trzeba zsumować i podzielić sumę przez ich liczbę.
Na początku zostanie odczytana liczebność zbioru **N** oraz dopuszczalne odchylenie **D**, wartość zaś sumy **S** wyzerowana. W trakcie kolejnego odczytywania dane dodawane są do wartości **S**. Po zakończeniu sumowania wartość **S** przerabiana jest na średnią. Teraz dane należy powtórnie przejrzeć, aby policzyć te, które mieszczą się w zadanym przedziale. Instrukcja **RESTORE 110** spowoduje, że najbliższa instrukcja **READ** odczyta pierwszą wartość z listy **DATA** w linii 110, pomijając linię 100. Zmienna **L**, wstępnie wyzerowana, odgrywa rolę licznika elementów należących do przedziału. Po wykryciu wśród danych wartości spełniającej założone warunki i mieszczącej się w przedziale, wartość **L** jest inkrementowana (zwiększana o 1). Po zakończeniu drugiego przeglądu danych wartość **L** odpowiada więc ilości szukanych liczb.

VAL **LET Y=VAL A$: LET U=VAL"123"**
LET Wynik=VAL "X*Y-SIN A"

Funkcja **VAL** dostarcza liczbowej wartości wyrażenia zapisanego w postaci tekstu. W większości innych dialektów języka BASIC argumentem **VAL** może być jedynie łańcuch przedstawiający stałą liczbową, np. "12.53". W ZX Spectrum argumentem **VAL** może być dowolne wyrażenie — podobnie jak miało to miejsce w instrukcji **READ**. Jeśli argument **VAL** jest wyrażeniem zawierającym nazwy funkcji i nazwy zmiennych, to funkcje zostaną wyliczone, a zamiast nazw zmiennych wstawione ich aktualne wartości.

STR$** **LET V$=STR$ X
LET W$=STR$(X/Y+10)+"stopni Celsjusza"

Funkcja **STR$** działa odwrotnie do **VAL**: dostarcza tekstu przedstawiającego

wartość argumentu **STR$**. Argument ten musi być liczbą. Funkcja **STR$** może być elementem wyrażeń łańcuchowych. Przyjmując, że wartość zmiennej **Cena** wyraża cenę produktu w złotówkach, równą np. 12 zł 30 gr, po uruchomieniu poniższego programu zmienna tekstowa **C$** otrzyma wartość: "12 zlotych 30 groszy".

Program 8

```
10 LET Cena=12.3
20 LET Zl=INT Cena
30 LET Gr=INT ((Cena-Zl)*100)
40 LET C$=STR$ Zl+" zlotych "+STR$ Gr+" groszy"
50 PRINT C$
```

Najpierw obliczana jest liczba pełnych złotówek. W tym celu wystarczy obciąć część ułamkową zmiennej **Cena** funkcją **INT**. Po odjęciu od pełnej ceny liczby złotówek otrzymujemy resztę, która pomnożona przez 100 daje liczbę groszy.
Liczba złotówek przekształcana jest na tekstowy zapis liczby, do którego doklejamy napis "złotych", za nim zaś — tekst przedstawiający liczbę groszy. Pozornie niepotrzebne użycie funkcji **INT** w linii **30** zapobiega sytuacji, w której wskutek ograniczonej dokładności przedstawiania liczb i nieuchronnych zaokrągleń, liczba groszy okazałaby się ułamkiem bardzo bliskim 30, np. **30.000001**. Usuńcie **INT** i obejrzyjcie efekty!

Organizacja ekranu

Na ekranie monitora lub zastępującego go telewizora wyróżniamy w ZX Spectrum trzy obszary. Obrzeże ekranu to tzw. ramka (BORDER). Użyteczny obszar ekranu to prostokąt podzielony umownie na 24 wiersze po 32 kwadratowe pola. W każdym polu może być umieszczony pojedynczy znak tekstu. Ogólna pojemność ekranu ZX Spectrum wynosi więc 24×32=768 znaków.
Wiersze numerowane są od 0 do 23. Wiersz nr 0 znajduje się u góry. Pola w wierszu numerowane są od lewej w zakresie 0—31. Obszar użyteczny podzielony jest umownie na dwa tzw. **okna** (WINDOW). Dwa najniższe wiersze o numerach 22 i 23 tworzą tzw. **okno systemowe**. Ukazują się w nim wszelkie komunikaty mikrokomputera — przede wszystkim informacja o błędach. Okno systemowe jest też obszarem roboczym przy wprowadzaniu danych.
Dwadzieścia dwie górne linie tworzą **okno użytkownika** (okno programowe). Jest ono naturalnym miejscem wyświetlania informacji dostarczanej przez działający program: wyników obliczeń, tablic, wykresów itd.

Dla każdego z 768 kwadratów można niezależnie ustalić kilka parametrów--**atrybutów**. Są to: kolor znaku, kolor tła, jaskrawość znaku i migotanie całego pola, polegające na periodycznej zamianie koloru tła i znaku. Do wyboru jest 8 barw, ponumerowanych od Ø (czarny) do 7 (biały). Numeracja jest tak dobrana, by na ekranie dwubarwnym barwom o niższym numerze odpowiadał ciemniejszy szary (wyższy numer = barwa jaśniejsza).
Każde pole złożone jest z 8 rzędów po 8 punktów. Każdemu z 64 punktów pola można niezależnie nadać kolor obowiązującego w danym polu tła lub kolor wypełnienia — atramentu. W pierwszym przypadku mówimy, że punkt jest skasowany, wygaszony, w drugim — wypełniony lub zapalony.
Operując grafiką, wygodnie jest widzieć ekran nie jako zbiór pól, lecz jako sieć pojedynczych punktów. W języku BASIC grafika dostępna jest tylko w oknie użytkownika. 22 linie po 8 rzędów dają w pionie rozdzielczość 176 punktów. Rozdzielczość w poziomie wynosi 32×8=256 punktów. Początek układu współrzędnych dla instrukcji graficznych — punkt o współrzędnych (Ø, Ø) — leży w lewym dolnym rogu okna użytkownika. Zestaw instrukcji oddziałujących na zawartość ekranu jest liczny.

BORDER **BORDER 3**
BORDER 2*K-7

Instrukcja **BORDER** ustala barwę ramki i kolor tła w oknie systemowym. Jej argumentem może być dowolne wyrażenie o wartości dopuszczalnej dla koloru. Ewentualne wartości ułamkowe zaokrąglane są do najbliższej liczby całkowitej (patrz instrukcja **GO TO**). W stosunku do ramki działanie instrukcji **BORDER** jest natychmiastowe. Na zmianę barwy tła w oknie systemowym trzeba czekać aż do pierwszego wyprowadzonego w nim komunikatu lub np. instrukcji wprowadzania danych **INPUT**.

PAPER, INK, BRIGHT, FLASH **INK 3 : PAPER 7-kol**
INK 8
BRIGHT 1 : FLASH Ø

Wszystkie instrukcje wyprowadzające informację na ekran zapalają lub gaszą jego wybrane punkty. Jeśli zmodyfikowany został punkt należący do jakiegoś pola, to zwykle ustawiane są także w całości lub w części atrybuty tego pola. Jeżeli w instrukcji piszącej lub rysującej na ekranie nie ma innych wskazówek, nadawane będą wartości atrybutów ustalone wymienionymi instrukcjami.
PAPER i **INK** określają obowiązujące barwy — odpowiednio tła (papieru) i wypełnienia (atramentu). Ich argumentem może być dowolne wyrażenie o wartości dopuszczalnej dla koloru. Dozwolone są też wartości 8 i 9. Np. **PAPER 8** oznacza, że w polach, w których wyprowadzana jest informacja, będzie zachowany dotychczasowy kolor tła. **INK 8** znaczy, że zapalane i gaszone będą tylko odpowiednie punkty, pozostanie jednak zachowana obowiązująca poprzednio w odpowiednich polach barwa atramentu.

PAPER 9 i **INK 9** sprawią, że komputer automatycznie dobierze taką wartość odpowiedniego atrybutu, aby zapewnić optymalny kontrast. Zasada automatycznego doboru atramentu stosowana jest standardowo w oknie systemowym.
BRIGHT 1 powoduje, że w polach, których treść modyfikowano, włączona zostanie podwyższona jaskrawość. **BRIGHT Ø** przywraca nadawanie normalnej jaskrawości. **BRIGHT 8** sprawia, że atrybut jaskrawości nie będzie modyfikowany, lecz pozostawiony we wszystkich polach na poprzednim poziomie. **FLASH 1**, **FLASH Ø** i **FLASH 8** oznaczają to samo w odniesieniu do atrybutu migotania.
Po włączeniu komputera wartości **PAPER**, **INK**, **BRIGHT** i **FLASH** ustawiane są automatycznie na 7, Ø, Ø, Ø.
Wykonanie instrukcji **PAPER 6** nie oznacza, że całe tło ekranu stanie się momentalnie żółte. Ustawianie atrybutów funkcjonuje jedynie w połączeniu z wyprowadzaniem informacji na ekran. Nowe atrybuty pojawią się tylko w tych polach ekranu, których zawartość zmodyfikowano.

INVERSE, OVER **INVERSE 1**
OVER Ø

INVERSE i **OVER** modyfikują nie atrybuty, lecz sposób zapalania i gaszenia punktów ekranu. PO **INVERSE 1** wszystkie punkty, które miałyby być zapalane, będą gaszone, i na odwrót. **INVERSE Ø** przywraca stan normalny. **OVER 1** sprawia, że wszystkie te punkty, które w normalnych warunkach miałyby być zapalone, zostaną przełączone w stan przeciwny do aktualnie zajmowanego. Jeśli po **OVER 1** miałby zostać zapalony punkt do tej pory skasowany, zostanie on zapalony normalnie. Gdyby jednak punkt ten był już wypełniony, to teraz zostanie skasowany. Dwukrotne wyprowadzenie tej samej informacji w trybie **OVER 1** w dowolne miejsce ekranu odtwarza więc w nim stan pierwotny. **OVER Ø** przywraca normalny tryb pracy.
Argumenty **INVERSE** i **OVER** mogą być oczywiście wyrażeniami liczbowymi. Reguły zaokrąglania — ogólnie przyjęte.

CLS **CLS**

Instrukcja **CLS** kasuje treść ekranu, gasząc wszystkie jego punkty i nadając wszystkim polom atrybuty ustawione uprzednio instrukcjami **PAPER**, **INK**, **BRIGHT**, **FLASH**. Z powodu wygaszania wszystkich punktów atrybut **INK** wydaje się być nieistotny. Może on jednak być potrzebny w trybie pracy bez modyfikacji atramentu (**INK 8**).

Wyprowadzanie informacji na ekran

Instrukcje wyświetlające informację na ekranie tworzą dwie grupy. Pierwsze, tekstowe, traktują ekran jako zbiór pól znakowych i działają na całe pola. Inne, graficzne, oddziałują tylko na pojedyncze punkty.

PRINT **PRINT "Wynik="; Y**
PRINT S, S/N

Instrukcja **PRINT** wyprowdza na ekran ciągi znaków. Mogą to być wartości wyrażeń liczbowych lub tekstowych. Pojedyncza instrukcja **PRINT** może wyprowadzić na ekran wiele elementów. Elementy te, umieszczone po rozkazie, tworzą tzw. listę elementów i mogą być dowolnymi, poprawnymi wyrażeniami tekstowymi i liczbowymi. Elementy listy muszą być rozdzielone separatorami: przecinkami lub średnikami. Jeśli separatorem jest średnik, następny element zostanie wyświetlony tuż za poprzednikiem, bez dodatkowego odstępu. Inaczej po przecinku. Ekran podzielony jest umownie na dwie pionowe kolumny-strefy, każda po 16 pól. Element po przecinku zostanie wyświetlony zawsze od początku nowej strefy. Jeśli przewidziane do wyświetlenia elementy nie mieszczą się w jednym wierszu, nastąpi automatyczne przejście do wiersza następnego.
Po wyświetleniu ostatniego elementu z listy nastąpi przejście do początku kolejnego wiersza. Wyjątkiem jest sytuacja, gdy listę zamyka samotny średnik lub przecinek. Jego obecność powstrzyma zmianę wiersza. Powstrzymując przejście do nowej linii, przecinek powoduje jednak zmianę strefy. Może być to równoważne zmianie wiersza. Najbezpieczniej używać średnika.
Gdy trzeba spowodować przejście do początku następnego wiersza bez wyprowadzania czegokolwiek, wystarczy samo puste **PRINT**. Chcąc spowodować przejście do nowego wiersza wewnątrz instrukcji **PRINT**, wystarczy na liście umieścić znak apostrofu " ' ".
Poniższy program wyświetla na ekranie trójkąt złożony z liter **"A"**. Na początku ekran jest kasowany, wyprowadzany jest nagłówek i ustawiane atrybuty. Pętle **FOR... NEXT** odliczają w kolejnych wierszach taką ilość "sklejonych" gwiazdek, ile wynosi numer wiersza. Puste **PRINT** powoduje przejście do budowy kolejnego "pięterka":

Program 9

```
10 CLS : PRINT "Oto trojkat",,''
20 INK 1: PAPER 6: BRIGHT 1
30 FOR I=1 TO 19
40   FOR K=1 TO I
50     PRINT "A";
60   NEXT K: PRINT
70 NEXT I
```

Instrukcja **PRINT** modyfikuje wszystkie atrybuty w danym polu, chyba że przy wybranych atrybutach użyto parametru 8. W powyższym przykładzie linie wyprowadzane były na ekran od góry do dołu, w naturalnej kolejności. W instrukcji **PRINT** użyć można przełączników logicznych, połączonych iloczynem logicznym z poprzedzającym je wyrażeniem tekstowym. Wyświetlone zostanie tylko to wyrażenie, którego przełącznik logiczny jest spełniony:

```
10 LET A$="JEDEN"
20 INPUT X
30 PRINT A$ AND X=1;"DWA" AND X=2
40 GO TO 20
```

Na liście **PRINT** można umieszczać modyfikatory, wskazujące bezpośrednio miejsce wyświetlenia następnego elementu.

TAB **PRINT TAB 10; Y TAB 20; Z**

Tabulator **TAB** powoduje wyświetlenie kolejnego elementu, począwszy od pola określonego wartością argumentu **TAB**. W powyższym przykładzie wartość zmiennej **Y** umieszczona zostanie począwszy od 10 pola, wartość Z — od 20 pola wiersza. Jeżeli aktualna pozycja w wierszu ma numer wyższy niż argument **TAB**, tabulacja jest ignorowana. Argument ułamkowy podlega zaokrągleniu.

AT **PRINT AT 10, 12 ; "SPEKTRUM"**

Modyfikator **AT** określa współrzędne pola ekranu, od którego rozpocząć się ma kolejny napis. Argumenty **AT** rozdzielone są przecinkiem i mogą być dowolnymi wyrażeniami liczbowymi (w razie potrzeby są zaokrąglane). Pierwszy argument przedstawia numer wiersza, drugi — numer pola (kolumny ekranu). W powyższym przykładzie napis **SPEKTRUM** zostanie wyświetlony w wierszu nr 10, poczynając od pola nr 12.

Modyfikator **AT**, podobnie jak **TAB**, może wystąpić wielokrotnie na liście **PRINT**, np.:

PRINT AT 0, 5 ;"TO"; AT 3, 8;"JEST"; AT 6, 11;"TEST"

W trakcie wykonywania instrukcji **PRINT** może zachodzić potrzeba częstych, lokalnych zmian atrybutów — np. przy wyprowadzaniu barwnych, migocących fragmentami napisów. Rozkazy **PAPER**, **INK**, **BRIGHT**, **FLASH**, **INVERSE**, **OVER** stosujemy wtedy nie jako samodzielne instrukcje, lecz lokalne modyfikatory na liście **PRINT**. Ustalona przez taki modyfikator wartość atrybutu obowiązuje wtedy co najwyżej do końca danej instrukcji **PRINT**. Po jej zakończeniu przywrócone zostają atrybuty ustalone przy włączeniu komputera lub zadane instrukcjami **PAPER**... **OVER**.

Program 10

```
10 CLS : PRINT "Modyfikatory lokalne"
20 PRINT AT 10,0; INK 2; PAPER 6;"TEST"; FLASH  1;
   "++++"; FLASH 0; BRIGHT 1;"JASNOSC"; PAPER 1; INK 4;
   "++++"; INVERSE 1;"INWERSJA"
30 PRINT AT 20,0;"Koniec testu modyfikatorow"
```

Na liście **PRINT** modyfikatory rozdzielamy średnikami lub przecinkami, nie dwukropkiem.

PLOT

```
PLOT INK 3 ; 100, 150
PLOT INVERSE 1 ; X, Y-50
```

Instrukcja **PLOT** zapala lub gasi punkt ekranu o wskazanych współrzędnych. Współrzędne mogą być zadane wyrażeniami liczbowymi — w razie potrzeby wartości są zaokrąglane. W przypadku wartości ujemnych uwzględniana jest ich wartość bezwzględna.
Wartość współrzędnych musi mieścić się w przedziale ⟨0, 255⟩ dla **X** i ⟨0, 175⟩ dla **Y**. Uwaga! Pierwsze wyrażenie podaje współrzędną **X**, drugie **Y** (odwrotnie niż po **AT**). Wyrażenia oddzielamy przecinkiem.
Po rozkazie **PLOT** wystąpić mogą, jak w **PRINT**, modyfikatory lokalne. Dozwolone są wszystkie modyfikatory z wyjątkiem **TAB** i **AT**. Działanie modyfikatorów po **PLOT** — podobnie jak w pozostałych instrukcjach graficznych — jest trochę inne niż w **PRINT**.
INVERSE i **OVER** włączają lub wyłączają poszczególne tryby na czas wykonywania danej instrukcji **PLOT**. Instrukcja:

```
PLOT INVERSE 1 ; 120, 30
```

powoduje np. wygaszenie punktu o współrzędnych (120, 30).
Jedynym atrybutem zmienianym normalnie przez instrukcję **PLOT** w kwadracie, do którego należy zapalany lub gaszony punkt, jest **INK**. Jeśli jednak po **PLOT** znajdą się modyfikatory **PAPER**, **BRIGHT** i **FLASH**, to zostaną zmodyfikowane także i odpowiadające im atrybuty. To samo dotyczy pozostałych instrukcji graficznych. Gdy chcemy, aby instrukcja graficzna nie zmieniała żadnych atrybutów, ograniczając się do samego zapalania i gaszenia punktów, wystarczy użyć **INK 8**.

CIRCLE

```
CIRCLE 128, 87, R
CIRCLE INK 8 ; INVERSE 1 ; X, Y, R+6
```

Instrukcja **CIRCLE** kreśli na ekranie okrąg, trzy jej argumenty wyznaczają położenie środka okręgu (X i Y) oraz promień. Argumenty mogą być zadane wyrażeniami liczbowymi o nieujemnych wartościach, okrąg musi całkowicie mieścić się na ekranie. Po **CIRCLE**, jak po **PLOT**, wystąpić mogą modyfikatory lokalne. Działają one identycznie jak w **PLOT**, dotycząc nie jednego, lecz wszystkich tworzących okrąg punktów.

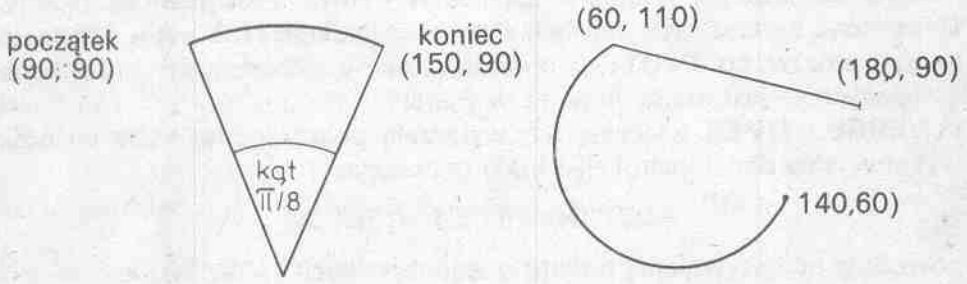
DRAW

```
DRAW -5, -35
DRAW INK kol ; 50, 0, PI/4
```

Instrukcja **DRAW** wykreśla odcinki prostej oraz odcinki okręgu (odcinek prostej może być uważany za odcinek okręgu o promieniu dążącym do nieskończoności). **DRAW** może mieć dwa lub trzy argumenty. Dwa pierwsze wyznaczają współrzędne końca odcinka względem jego początku. Począt-

kiem odcinka będzie ostatni punkt narysowany na ekranie poprzednią instrukcją **DRAW** lub **PLOT**. Zauważmy, że w odróżnieniu do **PLOT** i **CIRCLE**, wymagających współrzędnych bezwzględnych, **DRAW** akceptuje współrzędne względne.

Trzeci argument określa kąt, utworzony przez styczne do odcinka w jego punkcie początkowym i końcowym (inaczej: kąt między promieniami okręgu, wyznaczającymi końce odcinka — rys. 2). Kąt musi być podany w mierze łukowej (radianach), a jego wartość musi mieścić się w przedziale (−2Π, 2Π). Brak trzeciego argumentu oznacza przyjęcie zerowej wartości kąta, a więc wykreślanie odcinka prostej. Ten wariant instrukcji **DRAW** wykorzystywany jest najczęściej.



PLOT 90, 90:DRAW 60, 0, −PI/8

PLOT 180, 90: DRAW −120, 20
DRAW 80, −50, 1.3•PI

Rys. 2 Przykłady użycia instrukcji **DRAW**

Ujemna wartość kąta oznacza, że łuk od punktu początkowego do końcowego zataczany będzie zgodnie z kierunkiem ruchu wskazówek zegara. Przy kącie dodatnim łuk będzie zataczany w stronę przeciwną. Dopuszczalne modyfikatory i ich działanie: jak w **CIRCLE**.

POINT **LET P=POINT (128, X−1Ø)**

Funkcja **POINT** pozwala komputerowi "widzieć" ekran. Argumenty funkcji **POINT** muszą być ujęte w nawiasy, mogą być dowolnymi wyrażeniami i określają współrzędne wybranego punktu ekranu (w przypadku ujemnej wartości argumentu brana jest pod uwagę jego wartość bezwzględna — jak w **PLOT**). Jeśli wskazany punkt jest wypełniony (ma barwę atramentu), wartością funkcji będzie 1, w przeciwnym razie — Ø.

SCREEN$** **LET A$=SCREEN$(3, 2Ø)

Także i ta funkcja służy komputerowi do "oglądania" ekranu. **SCREEN$** traktuje jednak ekran jako zbiór wierszy i pól znakowych. Funkcja dostarcza jednoznakowego łańcucha, odpowiadającego zawartości wskazanego pola. W powyższym przykładzie jest nim dwudzieste pole wiersza nr 3. Gdy we

wskazanym kwadracie nie znajduje się odpowiednik żadnego ze znanych komputerowi znaków, **SCREEN$** dostarcza łańcuch pusty (może to mieć miejsce np. przy próbie odczytania fragmentu ekranu zapisanego instrukcjami graficznymi). Numer wiersza może być z zakresu Ø—23 — funkcja **SCREEN$** działa także w obszarze okna systemowego. Wadą **SCREEN$** jest niestety nierozpoznawanie dostępnych w Spectrum symboli graficznych ani znaków definiowanych przez użytkownika (UDG). Warto wiedzieć, że **SCREEN$** odczytuje także znaki zapisane inwersyjnie (po INVERSE 1), gdy punkty tworzące zarys znaku są wygaszone, pozostałe zaś punkty pola — zapalone (barwa atramentu).

ATTR **LET Atrybuty= ATTR(3, 2Ø)**

Funkcja **ATTR** uzupełnia **SCREEN$**. Akceptuje ona takie same argumenty. O ile jednak **SCREEN$** interesuje się tylko kompozycją zapalonych lub wygaszonych w danym kwadracie punktów, **ATTR** dostarcza wyłącznie informacji o obowiązujących w danym polu atrybutach. Wartość **ATTR** jest liczbą z zakresu Ø—255, zawierającą w zakodowanej formie informacje o kolorach tła, wypełnienia, jaskrawości i włączeniu migotania.

Wartość **ATTR** zależy od wartości atrybutów w danym polu w następujący sposób:

wartość **ATTR = FLASH×128 + BRIGHT×64 + PAPER×8 + INK**

INPUT

```
INPUT X, A$
INPUT AT Ø, 5;"Imie:"; I$, AT 1, 6;"Wiek:"; W
INPUT LINE "Pelny adres:"; A$
```

Instrukcja **INPUT** powoduje przejęcie przez komputer określonej informacji z klawiatury. Omawianie **INPUT** w tym momencie może wydać się dziwne. Instrukcja ta korzysta jednak z wielu własności instrukcji **PRINT**. Działanie **INPUT** jest podobne do **READ**. Wartości nie są jednak pobierane ze zbioru **DATA**. Komputer oczekuje na podanie danych z klawiatury. Podobnie jak **READ**, **INPUT** akceptuje także wyrażenia liczbowe i tekstowe. Dla wyjaśnienia użytkownikowi, na jaką informację komputer oczekuje, można przed nazwą zmiennej umieścić stosowny komentarz-podpowiedź w postaci stałej tekstowej (tekstu w cudzysłowach). Podpowiedź pojawi się w oknie systemowym przy wprowadzaniu wartości zmiennej. W oknie systemowym pojawia się też kursor, informujący o oczekiwaniu na daną. Gdy komputer oczekuje na liczbę, kursor występuje samodzielnie. Oczekiwanie na tekst sygnalizowane jest kursorem w znakach cudzysłowu.

Wyprowadzając podpowiedź wolno użyć wszystkich modyfikatorów lokalnych, dopuszczalnych w **PRINT**. Stosując **AT** trzeba jednak pamiętać, że w instrukcji **INPUT** obowiązuje numeracja wierszy od góry okna systemowego. Linia 22 ma więc w oknie systemowym numer Ø, linia 23 — nr 1. Modyfikator **AT** z pierwszym argumentem większym od 1 spowoduje powiększenie okna systemowego kosztem okna użytkownika.

Wprowadzając zmienne tekstowe należy pamiętać, że tekst nie może zawierać znaków cudzysłowu. Niekiedy, np. przy redagowaniu tekstów, zaistnieć może potrzeba wczytania z klawiatury tekstów zawierających cudzysłowy. W tym wypadku nazwę wprowadzanej zmiennej poprzedzamy słowem kluczowym **LINE**.

```
INPUT "Numer="; Nr , "Nazwa:"; LINE N$ ,"Rok:"; R
```

W trybie **LINE**, mimo wprowadzania tekstu, kursor występuje bez cudzysłowów.
Przerwać program w trakcie wykonywania **INPUT** można wprowadzając słowo kluczowe **STOP** i wciskając **ENTER**. Jeśli oczekiwana była zmienna tekstowa, trzeba przedtem skasować lewy cudzysłów, tak by **STOP** był pierwszym elementem w linii. W trybie **LINE** opisana metoda zawodzi. Komputer akceptuje wszystkie symbole, nie wyłączając słowa kluczowego **STOP**. Jedynym wyjściem jest użycie symbolu "kursor w dół" — równocześnie wciskając **CAPS SHIFT** i klawisz z cyfrą 6.

Struktury danych: tablice i łańcuchy

ZX Spectrum traktuje łańcuchy w zasadzie jako tablice złożone z pojedynczych znaków. Dotyczy to nawet zmiennych tekstowych prostych. W odróżnieniu od większości innych dialektów, najmniejszą dopuszczalną wartością wskaźników tablicy (indeksów) jest w Spectrum nie Ø, lecz 1. Nazwa tablicy musi być jednoliterowa. W programie mogą wystąpić zmienne liczbowe proste i tablice liczbowe o identycznych nazwach. W przypadku tablic tekstowych i zmiennych tekstowych prostych jest to jednak zabronione.

DIM

```
DIM a(1Ø)
DIM B$(2Ø, 5)
```

Instrukcja **DIM** deklaruje tablicę, rezerwując niezbędną pamięć. Każda tablica przed pierwszym użyciem musi być w Spectrum zadeklarowana — nie ma tu automatycznej deklaracji z chwilą pierwszego użycia elementu, jak w innych dialektach. Jedną instrukcją **DIM** można zadeklarować tylko pojedynczą tablicę. Dozwolony gdzie indziej zapis:

```
DIM B(15), X (3, 3Ø)
```

rozbić trzeba na dwie instrukcje:

```
DIM B(15) : DIM X(3, 3Ø)
```

Po zadeklarowaniu tablicy liczbowej wszystkie jej elementy mają wartość Ø. Elementy tablic tekstowych mają jednakową, sztywną długość. Jest to widoczne już w deklaracji. Instrukcja:

```
DIM T$(1Ø, 2Ø)
```

deklaruje tablicę tekstową złożoną z 10 elementów, każdy długości dokładnie 20 znaków. Ostatnie wyrażenie w deklaracji tablicy tekstowej określa zawsze długość pola pojedynczego elementu.
Występuje tu istotna różnica w stosunku do innych dialektów, gdzie elementem tablicy może być łańcuch zmiennej długości, a po zadeklarowaniu tablicy wszystkim jej elementom przypisywane są łańcuchy puste, tzn. łańcuchy długości 0 znaków. W Spectrum wszystkie pola zadeklarowanej tablicy wypełnione są spacjami. Wynika z tego m.in. fakt, że element tablicy nie może być nigdy łańcuchem pustym.
W Spectrum można zadeklarować też zmienną tekstową prostą sztywno ustalonej długości. Zadeklarowana w ten sposób zmienna prosta zachowuje się dokładnie tak, jak element tablicy długości, powiedzmy, 30 znaków:

DIM X$ (30)

Nie zadeklarowana w **DIM** zmienna tekstowa prosta może mieć natomiast długość praktycznie dowolną.
Operacje na zmiennych tekstowych ułatwia szereg funkcji:

LEN **LET Y=LEN A$**

Funkcja **LEN** wyznacza długość łańcucha, będącego jej argumentem. Łańcuch ten może być zadany dowolnym wyrażeniem łańcuchowym. Użycie **LEN** jest sensowne tylko w przypadku nie deklarowanych zmiennych tekstowych prostych:

Program 11

```
10 DIM X$(2,20)
20 LET A$="Delta": LET X$(1)="ABCD"
30 PRINT LEN A$: PRINT LEN X$(1),LEN X$(2)
```

Najpierw deklarujemy tablicę tekstową o dwóch elementach po 20 znaków. Zmiennej tekstowej **A$** przypisujemy pięcioznakowy tekst **"Delta"**, a pierwszemu elementowi tablicy **X$** — czteroliterowy tekst **"ABCD"**. W przypadku **A$** funkcja **LEN** dostarczy poprawnej długości, równej 5. Obydwie pozostałe długości wyniosą 20, mimo że w pierwszym przypadku oczekiwalibyśmy raczej 4, a w drugim 0 (łańcuch pusty), jak w innych maszynach.
Jeżeli przypiszemy elementowi tablicy tekst krótszy od zarezerwowanego pola, zostanie on uzupełniony spacjami do przepisanej długości. Próba przypisania elementowi tablicy tekstu o większej liczbie znaków niż zadeklarowana długość elementu spowoduje obcięcie nie mieszczących się znaków. Ilustruje to przykład:

Program 12

```
10 DIM C$(2, 5)
20 LET C$(1)="KOT"
30 LET C$(2)="ALFABET"
40 LET A$=C$(1)+C$(2): PRINT A$
```

Po uruchomieniu programu na ekranie pojawi się napis: **KOT ALFAB**. W pierwszym elemencie do słowa **"KOT"** dodano dwie spacje, uzupełniając tekst do długości 5 znaków. Łańcuch C$(2) przyjął tylko pięć pierwszych znaków tekstu **"ALFABET"**.
Łańcuchy sztywnej długości powodują nieekonomiczne wykorzystanie pamięci w przypadkach, gdy w jednej tablicy gromadzimy teksty (słowa, zdania itp.) znacznie różniących się długością. Trzeba bowiem wtedy przy deklaracji tablicy uwzględniać największą możliwą długość elementu. Poważne komplikacje wystąpią też przy próbach porównania elementów tablic tekstowych ze stałymi tekstowymi lub tekstowymi zmiennymi prostymi. Oto przykład:

Program 13

```
10 DIM X$(3, 10)
20 LET X$(1)="alfa"
30 LET X$(2)="DELTA"
40 LET X$(3)="Sprawdzian"
50 LET W$="DELTA"
60 IF X$(1)="alfa" THEN PRINT "1 OK"
70 IF X$(2)=W$ THEN PRINT     "2 OK"
80 IF X$(3)="Sprawdzian" THEN PRINT "3 OK"
```

Na ekranie pojawi się tylko napis **"3 OK"**. Jedynie w instrukcji 80 długość elementu tablicy i długość stałej tekstowej "Sprawdzian" były zgodne — po 10 znaków. Zachodziła też zgodność odpowiednich pozycji obu łańcuchów. W pozostałych przypadkach czteroznakowy tekst **"alfa"** i pięcioznakowy **"DELTA"** porównywane były z dziesięcioznakowymi łańcuchami: "alfa _____" i "**DELTA** _____" (znak "_" symbolizuje spację).
Prostym, lecz nieekonomicznym, nieeleganckim i nie zawsze możliwym wyjściem jest dopełnienie w przeznaczonych do porównywania z elementami tablicy wzorcach tekstów spacjami do zadanej długości. W programie 13 wystarczyłoby np. zmodyfikować linie 50 i 60:

```
50 LET W$= "DELTA     "
60 IF X$(1)="alfa     " THEN PRINT "1 OK"
```

Lepszym rozwiązaniem jest skorzystanie z pośrednictwa zmiennych prostych ustalonej długości, odpowiadającej długości elementów tablicy. W programie 13 wystarczyłoby dopisać:

15 DIM W$(1Ø)

Metodę tę wykorzystano w prostym programie trenującym znajomość słówek angielskich:

Program 14

```
 10 DIM A$(100,15): DIM P$(100,15)
 20 DIM Z$(15): CLS
 30 READ IloscSlowek
 40 FOR I=1 TO IloscSlowek
 50   READ A$(I), P$(I)
 60 NEXT I
 70 LET K=INT (RND*2): LET N=INT (IloscSlowek*RND)+1
 80 IF K=0 THEN LET X$=A$(N): LET Y$=P$(N)
 90 IF K=1 THEN LET X$=P$(N): LET Y$=A$(N)
100 PRINT INVERSE K; X$; "=";
110 INPUT "Podaj slowko: "; Z$
120 PRINT INVERSE K=0; FLASH Z$<>Y$; Y$
130 GO TO 70
140 DATA 8
150 DATA "ink","atrament","paper","papier"
160 DATA "copybook","zeszyt","clock","zegar"
170 DATA "keyboard","klawiatura","time","czas"
180 DATA "defense","obrona","sword","miecz"
```

Spośród obu zadeklarowanych tablic — **A$** przeznaczona jest na słówka angielskie, **P$** — na polskie. Maksymalną długość słówka oszacowaliśmy na 15 znaków. Założyliśmy też, że nasz słownik może przechowywać maksymalnie 100 par słówek. Parametry te można oczywiście zmienić, inaczej deklarując tablice. Zmienna **Z$**, sztywnej długości 15 znaków, będzie przechowalnią słowa aktualnie porównywanego z elementem którejś z tablic. Słówka umieszczone są parami w zbiorze **DATA**. Pierwsza linia zawiera informację o dostępnej aktualnie w zbiorze liczbie par słówek. Program najpierw odczytuje liczbę par i przypisuje ją zmiennej **IloscSlowek**. Następnie, w pętli **FOR... NEXT**, zapełniane są słówkami obydwie tablice.
Właściwe działanie "trenera" rozpoczyna się w linii 7Ø. Zmienna **K** przyjmuje przypadkowo z teoretycznie równym prawdopodobieństwem wartość Ø lub 1. Wartość Ø oznacza, że w pytaniu przedstawione będzie słówko angielskie, a zadaniem egzaminowanego będzie podać polski odpowiednik. Dla **K**=1 będzie odwrotnie. Zmiennej **X$** przypisuje się pytanie, **Y$** — poprawną odpowiedź (linie 8Ø, 9Ø). Zanim zadamy pytanie o słówko, musimy jeszcze

wylosować któreś ze słówek zawartych w tablicy. "Maszyna losująca" zlokalizowana jest w drugiej instrukcji linii 70. Zmienna **N** przyjmuje losową wartość przedziału **1 — IloscSlowek**. Po wybraniu słówek i przypisaniu ich zmiennym **X$**, **Y$** komputer wyświetla na ekranie słówko stanowiące treść pytania. Słowa angielskie wyświetlane są normalnie, polskie — inwersyjnie. Ułatwia to orientację.
Po zadaniu pytania komputer oczekuje na odpowiedź (linia 110). Gdy ćwiczący poda słówko, komputer obok słówka zadanego wyświetli poprawną odpowiedź: angielską normalnie, polską inwersyjnie. Zapewnia to wyrażenie logiczne **K=0** umieszczone po słowie **INVERSE**.
W instrukcji **PRINT** zauważamy modyfikator **FLASH**. Jego argumentem jest z kolei warunek **Z$<>Y$**. Po podaniu poprawnej odpowiedzi warunek nie będzie spełniony, więc **FLASH** nie zostanie uaktywniony. Przy odpowiedzi błędnej warunek jest spełniony, jego arytmetyczna wartość wynosi 1. Migotanie zostanie włączone. Poprawna odpowiedź podana przez komputer będzie więc migotać, zwracając uwagę na popełniony błąd.
Przedstawiony program można zastosować oczywiście do dowolnych ćwiczeń lingwistycznych. Wystarczy po prostu wymienić zbiór danych. W zastosowaniach praktycznych liczba słówek powinna wynosić co najmniej kilkaset.
Operowanie fragmentami łańcuchów jest w Spectrum także niestandardowe. Aby pobrać z łańcucha dowolny znak, wystarczy wskazać go indeksem, jak element tablicy:

Program 15

```
10 LET R$="PRZYKLAD"
20 LET A$=R$(5)
30 LET B$="SPECTRUM"(7)
```

Łańcuchowi **A$** przypisana zostanie litera **"K"**, łańcuchowi **B$** — litera **"U"**.
Do wybierania wycinków zmiennej długości służą w innych systemach zazwyczaj specjalne funkcje (zwykle **LEFT$**, **MID$**, **RIGHT$**).
W Spectrum wystarczy do łańcucha źródłowego (tego, z którego wybieramy fragment) dołączyć specyfikację żądanego wycinka. Wskazuje ona, od którego do którego znaku należy fragment skopiować. Przykład:

Program 16

```
10 DIM A$(10, 10)
20 LET A$(1)="ZESTAW LITER"
30 LET B$="abcdefghijkl"
40 LET C$=B$(3 TO 7)
50 LET D$=A$(1)(5 TO 10)
```

```
60 LET E$=B$(8 TO )
70 LET F$=A$(1)( TO 5)
80 LET H$="NAPIS"(2 TO 4)
90 PRINT C$, D$, E$, F$, H$
```

Po uruchomieniu programu ekran wypełni się następującymi napisami:

```
cdefg          AW LIT
hijkl          ZESTA
API .
```

Jak widać w powyższym przykładzie, zamiast **(1 TO X)** można w uproszczeniu zapisać: **(TO X)**. Podobnie w sytuacji, gdy interesują nas wszystkie znaki począwszy np. od szóstego do końca, wystarczy podać **(6 TO)**.
Specyfikację podłańcucha (fragmentu) stosować można także po lewej stronie znaku przypisania. Innymi słowy, chcąc zmodyfikować tylko fragment jakiegoś łańcucha, nie trzeba przepisywać go w całości:

Program 17

```
10 LET A$="OCZYSZCZONO PLAC BUDOWY"
20 LET A$(1 TO 11)="ZREWIDOWANO"
30 LET A$(16)="N"
40 PRINT A$
```

Po uruchomieniu wyświetli się napis: **"ZREWIDOWANO PLAN BUDOWY"**. Pamiętajmy, że znaki wymieniać można wyłącznie w stosunku jeden za jeden. Próba przypisania:

LET A$ ="ABCDEFGH" : LET A$(4 TO 7)="X"

nie utworzy łańcucha **"ABCXH"**, lecz **"ABCX H"**. Łańcuch **"X"** został uzupełniony do wymaganej długości spacjami. Pdobnie przypisanie:

LET B$ ="ABCDEFGH" : LET B$(3 TO 4)="+++++"

da łańcuch **"AB++EFGH"** zamiast spodziewanego **"AB++++EFGH"**.
W pracy z łańcuchami wykorzystujemy jeszcze inne funkcje:

CODE

```
LET X=CODE "A"
LET Y=CODE A$
```

Każdy znak alfabetu, symbol graficzny itp. przedstawiony jest we wnętrzu Spectrum jako niewielka liczba całkowita. Funkcja **CODE** dostarcza liczby, będącej kodem pierwszego znaku w łańcuchu występującym jako argument. Gdy łańcuch jest pusty, wartością **CODE** jest Ø. Odpowiednikiem **CODE** w innych mikrokomputerach jest funkcja **ASC**.

CHR$

```
LET A$=CHR$ 65
PRINT CHR$(48+X)
```

Funkcja **CHR$** działa odwrotnie do **CODE**. Dostarcza jednoznakowego łańcucha, odpowiadającego w wewnętrznym kodzie komputera liczbie będącej argumentem funkcji. Liczba musi mieścić się w zakresie Ø—255. Funkcji **CHR$** użyć można np. do tworzenia znaków (elementów łańcucha), których nie ma na standardowej klawiaturze ZX Spectrum.

INKEY$

```
LET A$ =INKEY$
IF INKEY$="V" THEN GO TO 2Ø
```

Funkcja **INKEY$** testuje klawiaturę. Jeśli w chwili wywołania nie był wciśnięty żaden klawisz, wartością funkcji jest łańcuch pusty. W przeciwnym wypadku wartością funkcji jest pojedynczy znak, odpowiadający naciśniętemu aktualnie klawiszowi.
W odróżnieniu do **INPUT**, **INKEY$** dostarcza informacji natychmiast, nie czekając na **ENTER**. Po wciśnięciu klawisza odpowiadający mu znak nie pojawia się na ekranie w oknie systemowym. **INKEY$** jest przydatna przy tworzeniu własnych systemów wprowadzania danych, zwłaszcza tam, gdzie wymagana jest duża szybkość reakcji — np. przy grach.

Inne instrukcje ZX Spectrum

NEW

```
NEW
```

NEW kasuje zawarty w pamięci program i wszystkie zmienne. Program i zmienne likwidowane są fizycznie, tzn. zajęty przez nie obszar pamięci zapisywany jest nową zawartością.

CLEAR

```
CLEAR
CLEAR 3ØØØØ
```

Instrukcja **CLEAR** bez argumentu likwiduje wszystkie zadeklarowane do tej pory tablice i zmienne proste, zwalniając zajmowaną przez nie pamięć. Jeśli **CLEAR** posiada argument liczbowy, to oprócz zerowania zmiennych określany jest najwyższy adres pamięci RAM, dostępny dla interpretera języka **BASIC**. Numer najwyższej dostępnej komórki (adres komórki) równy jest argumentowi instrukcji.

RANDOMIZE

```
RANDOMIZE
RANDOMIZE 516+X
```

Instrukcja **RANDOMIZE** wpływa na pracę generatora liczb losowych, uruchamianego funkcją **RND**. Generowana liczba jest właściwie pseudo-przypadkowa, gdyż powstaje według założonego przepisu — algorytmu. **RND** dostarcza więc w gruncie rzeczy powtarzalnych serii liczb, nawet jeśli serie te są długie i dobrze imitują liczby losowe.

Po **RANDOMIZE** bez argumentu (znaczy to samo, co **RANDOMIZE Ø**) seria liczb pseudolosowych startuje w miejscu zupełnie przypadkowym, niemożliwym do przewidzenia. Przy niezerowym argumencie generator liczb losowych wraca do ściśle określonego stanu, jednoznacznie sprecyzowanego wartością argumentu. Może być to przydatne np. podczas uruchamiania programów, do czego potrzebny jest stale ten sam ciąg liczb pseudolosowych. Przed każdym uruchomieniem programu wystarczy wykonać instrukcję **RANDOMIZE** z ustalonym argumentem. Oto przykład:

Program 18

```
10 RANDOMIZE 456
20 FOR I=1 TO 22
30   PRINT RND
40 NEXT I
```

Uruchamiając powyższy program wielokrotnie, uzyskamy na ekranie ciąg liczb wyglądających na losowe. Za każdym razem będą to jednak te same liczby. Gdy zmienimy argument **RANDOMIZE**, na ekranie pojawi się co prawda inny zestaw liczb, jednak i on będzie się powtarzał po każdym uruchomieniu. **RANDOMIZE Ø** (bądź samo **RANDOMIZE**) dostarczy za każdym uruchomieniem zupełnie innego zbioru wartości.

REM **REM To jest komentarz**

Instrukcja komentarza **REM** służy głównie do zamieszczania objaśnień w treści programu. Komentowanie postaci danych, przeznaczenia zmiennych, funkcji poszczególnych grup instrukcji jest dobrym zwyczajem. Po **REM** może być umieszczony zupełnie dowolny zbiór znaków. Wykonując program, komputer przeskakuje instrukcję **REM**.

BEEP **BEEP 2, 1**

Rozkaz **BEEP** wytwarza dźwięki o zadanej wysokości i czasie trwania. Pierwszy z dwóch argumentów określa czas trwania dźwięku w sekundach. Drugi określa wysokość tonu. Dozwolone są wartości od –60 do +69. Wartość Ø odpowiada tonowi C (261.63 Hz). Od tego tonu rozciąga się w dół i w górę numeracja pozostałych dźwięków w odstępach półtonowych:

Ø C
1 Cis
2 D
. .
. .
. .
12 C
13 Cis

itp. Jak widać, dwanaście tonów tworzy oktawę. Podobnie interpretuje się wartości ujemne. –12 oznacza "głębokie" C.
Generacja dźwięku odbywa się całkowicie programowo. Wytwarzanie dźwięku absorbuje procesor całkowicie. Wstrzymywane jest nawet odliczanie czasu i testowanie klawiatury (nie działa np. klawisz **BREAK**).

PAUSE **PAUSE 5ØØ**
PAUSE Ø

Instrukcja **PAUSE** wstrzymuje wykonanie programu na zadany czas. Argument określa czas trwania pauzy w pięćdziesiątych częściach sekundy. Instrukcja **PAUSE 15Ø** powoduje więc zwłokę trzysekundową. Jeśli w trakcie wykonywania instrukcji **PAUSE** zostanie naciśnięty jakikolwiek klawisz, pauza zostanie natychmiast przerwana. Jeśli natomiast chcemy, by program odczekał zadany czas bez względu na manipulacje przy klawiaturze, musimy posłużyć się opóźniającą pętlą **FOR... NEXT**, np.:

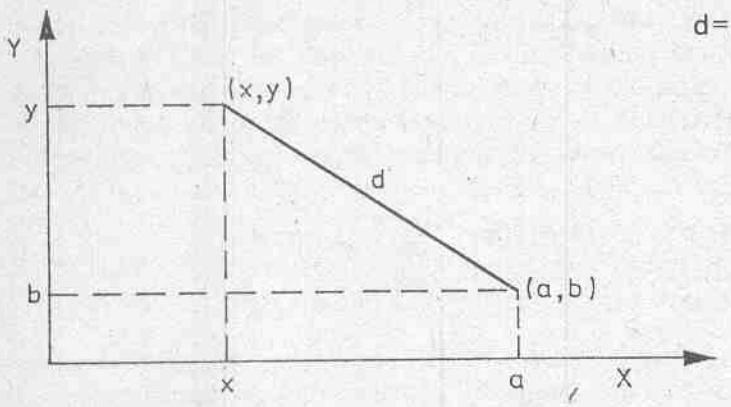
FOR T=1 TO 5ØØ : NEXT T

Wartość opóźnienia dobieramy eksperymentalnie, zmieniając zakres zmian zmiennej sterującej.
Instrukcja **PAUSE Ø** powoduje pauzę o nie określonej długości. Wznowienie pracy programu nastąpi dopiero po naciśnięciu dowolnego klawisza.

DEF FN **DEF FN Q(A)= A*A+5**
DEF FN B$(A$,V)=CHR$ V+A$(2 TO LEN A$–1)+CHR$ V

DEF FN pozwala użytkownikowi na definiowanie w razie potrzeby własnych funkcji. Funkcje mogą być liczbowe lub łańcuchowe, jednej lub wielu zmiennych. Możliwe jest też tworzenie funkcji bez argumentów. Aby zdefiniować funkcję, należy po słowie kluczowym **DEF FN** umieścić jej nazwę, uzupełnioną ewentualnie znakiem $ w przypadku funkcji tekstowej. Nazwa funkcji musi być jednoliterowa. Uniemożliwia to niestety nadawanie funkcjom sugestywnych, mnemonicznych nazw. Za nazwą należy w parze nawiasów wymienić wszystkie tzw. parametry formalne funkcji. Jeśli funkcja nie ma parametrów formalnych, trzeba pozostawić pustą parę nawiasów. Nazwy parametrów formalnych także muszą być jednoliterowe.
Po prawej stronie symbolu = przedstawiamy właściwy przepis na wyznaczenie wartości funkcji. Tam, gdzie w przepisie występują parametry formalne, po wywołaniu funkcji podstawione zostaną odpowiednie parametry aktualne. Parametry formalne są więc jedynie znacznikami miejsca podstawienia parametrów aktualnych i nie mają nic wspólnego z użytymi w programie zmiennymi o takiej samej nazwie.
Aby wywołać w programie funkcję, wystarczy umieścić w wyrażeniu słowo kluczowe **FN** z następującą po nim nazwą funkcji. Za nazwą funkcji umieszczamy w nawiasach parametry aktualne. Mogą one być zadane w postaci wyrażeń. I tu przykład: zdefiniujemy funkcję wyznaczającą

odległość dwóch punktów na płaszczyźnie. Argumentami funkcji będą współrzędne tych punktów. Znany z geometrii wzór mówi, że odległość punktów równa się pierwiastkowi kwadratowemu z kwadratów różnic odpowiednich współrzędnych tych punktów (rys. 3).



$$d=\sqrt{(x-a)^2+(y-b)^2}$$

Rys. 3

Oznaczmy współrzędne pierwszego punktu przez **x,y**, drugiego przez **a,b**.

Program 19

```
10 DEF FN D(x,y,a,b)=SQR ((x-a)*(x-a)+(y-b)*(y-b))
20 LET X1=3: LET Y1=9: LET X2=8: LET Y2=-2
30 LET Z=FN D(X1,X2,Y1,Y2)
40 LET W=FN D(X1+5,23,20*SIN 0.7,Y2)
50 PRINT Z,W,10+FN D(0,0,45,50)
```

Po wywołaniu funkcji najpierw obliczone będą wartości wszystkich parametrów aktualnych, podanych w nawiasach za nazwą funkcji. Parametry aktualne mogą być dowolnymi wyrażeniami, odwołującymi się tak do funkcji standardowych, jak i zdefiniowanych, a nawet do funkcji właśnie obliczanej. Ilość i charakter (liczbowy, tekstowy) parametrów aktualnych odpowiadać muszą ściśle podanym w definicji parametrom formalnym.
Po obliczeniu argumentów wartości wyrażeń zostaną wstawione do zdefiniowanego w **DEF FN** wzoru w miejsce odpowiednich parametrów formalnych. Przykładem funkcji bez parametrów może być funkcja losowa, która z równym prawdopodobieństwem przyjmować ma wartości będące liczbami całkowitymi z zakresu 1—6, jak przy grze w kości:

DEF FN R()=INT(6*RND) +1

Definiując ją, skorzystaliśmy z funkcji standardowej **RND**. Funkcja ta przyjmuje jednak wartości nie całkowite, lecz rzeczywiste i to mniejsze od 1. Mnożąc wartość **RND** przez 6 uzyskujemy wartości losowe z przedziału

Ø—5.9999999. Po odcięciu części ułamkowej mamy liczbę całkowitą od Ø do 5. Teraz wystarczy dodać 1.
Możliwość definiowania rozbudowanych funkcji wielu zmiennych jest w ZX Spectrum mocnym narzędziem programistycznym. Będziemy z niej intensywnie korzystać. Na razie dla treningu zdefiniujemy funkcje **FN L$**, **FN R$** i **FN M$**, będące odpowiednikami występujących w innych dialektach funkcji **LEFT$**, **RIGHT$** i **MID$**, a służących do wycinania podłańcuchów. Pierwszym argumentem każdej funkcji jest łańcuch źródłowy, a ostatnim — liczba znaków do wycięcia. W funkcji **LEFT$** znaki wybierane są od lewego, w funkcji **RIGHT$** — od prawego końca łańcucha źródłowego. W **MID$** występuje trzeci, środkowy parametr, określający, od którego znaku rozpocząć wycinanie. Tak np.:

LEFT$ ("ABCDEFG", 4)="ABCD"
RIGHT$("JKLMNO", 2)="NO"
MID$ ("FUNKCJA", 3, 4)="NKCJ"

Teraz zdefiniujemy odpowiednie funkcje dla Spectrum. Tak zdefiniowane funkcje mogą przydać się przy adaptacji dla Spectrum programów napisanych w MICROSOFT-BASIC, np. dla MERITUM lub C-64.

```
10        DEF FN L$(A$, D) = A$( TO  D)
20        DEF FN R$(A$, D) = A$(LEN A$-D+1 TO )
30        DEF FN M$(A$,P,D)= A$(P TO P+D-1)
```

Współpraca z magnetofonem jako pamięcią zewnętrzną

Na taśmie magnetofonowej można zapisywać w ZX Spectrum tylko spójne obszary pamięci. Obszar taki tworzy m.in. tekst programu wraz z obszarem danych. Aby go zapisać, wystarczy zlecenie **SAVE "nazwa"**. Jeśli chcemy, aby program po ponownym załadowaniu automatycznie się uruchomił, dodajemy po nazwie słowo **LINE** z numerem wiersza, od którego nastąpić ma start programu, np.:

SAVE "Test" LINE 2Ø

Po załadowaniu takiego programu instrukcją **LOAD "Test"** lub **LOAD " "** nastąpi skok do wskazanej linii 2Ø. Skok ten jest odpowiednikiem instrukcji **GO TO 2Ø**, a nie **RUN 2Ø**! To bardzo ważne, bowiem program zapisywany jest na taśmę wraz ze wszystkimi zmiennymi, tablicami itp. Tak więc instrukcja **SAVE** powoduje jakby zamrożenie programu w określonym stanie. Po powtórnym załadowaniu jesteśmy w stanie "reanimować" zakonserwowany program i kontynuować jego pracę. Tam, gdzie chcemy uniknąć

niepotrzebnego zapisywania zmiennych (ich objętość może niekiedy wielokrotnie przewyższać objętość programu), należy przed **SAVE** usunąć zmienne instrukcją **CLEAR**.

MERGE **MERGE "Czesc 2"**

Przy tworzeniu programów o większej objętości i o modułowej strukturze instrukcja **MERGE** jest wielką pomocą. Umożliwia ona oddzielne programowanie i testowanie poszczególnych modułów. Instrukcja **MERGE** jest zbliżona do **LOAD**. O ile jednak **LOAD** niszczy zawarty dotychczas w pamięci program i jego zmienne, to **MERGE** powoduje dołączenie ładowanego programu do programu zawartego już w pamięci operacyjnej. Obowiązują przy tym takie same reguły, jak przy wprowadzaniu programu z klawiatury. Jeśli linia o danym numerze występuje zarówno w programie dotychczas zawartym w pamięci, jak i we fragmencie dołączanym z taśmy przez **MERGE**, to linia zawarta w pamięci zostanie usunięta i zastąpiona przez linię wczytaną z taśmy. Może się zdarzyć, że po dołączeniu instrukcją **MERGE** programu z taśmy całkowita długość programu ulegnie zmniejszeniu!

Przy wykonywaniu instrukcji **MERGE** cały dołączany moduł programowy ładowany jest najpierw do pamięci mikrokomputera, a dopiero potem następują właściwe manipulacje w programie. Zapotrzebowanie na pamięć w trakcie wykonywania instrukcji **MERGE** jest więc znacznie większe, niż wynika to z ostatecznej objętości programu. Aby uniknąć przepełnienia pamięci, celowe jest dołączanie krótkich modułów. Jeśli mamy do dołączenia długi moduł, można go rozbić na kilka fragmentów, dołączanych kolejnymi instrukcjami **MERGE**.

Instrukcja **MERGE** zachowuje pierwotne wartości zmiennych w programie. Dzieje się to jednak tylko wtedy, gdy dołączany program został zapamiętany bez zmiennych, tzn. przed **SAVE** wykonano **CLEAR**. W przeciwnym razie zmienne występujące tylko w dołączanym fragmencie zostaną dopisane do zawartych już w pamięci zmiennych razem z wartościami, zmienne zaś o nazwach już występujących w poprzednim programie zostaną zastąpione — podobnie jak linie programu.

W wielu zastosowaniach interesuje nas oddzielne zapamiętanie programu i danych. ZX Spectrum pozwala zapamiętywać na taśmie jedynie tablice, i to w całości. Ponieważ zmienne tekstowe traktowane są jak tablice, także i je można zapisać w pamięci magnetofonowej (jeśli zostały zadeklarowane):

Program 20

```
10 DIM A(10,10): DIM X$(30,15): DIM W$(20)
20 LET W$="TO MA BYC ZAPISANE"
30 SAVE "Tab.A" DATA A()
40 SAVE "TABLICA X$" DATA X$()
50 SAVE "Dane" DATA W$()
```

Zauważmy, że za każdą nazwą tablicy, a nawet za deklarowaną zmienną łańcuchową prostą(!) następować musi pusta para nawiasów. Nazwa zapisanego na taśmie zbioru: programu lub danych może być złożona najwyżej z 10 znaków.

Ładujemy dane z taśmy instrukcją: **LOAD "nazwa" DATA nazwa zmiennej ()**. Jeśli ładujemy tablice z taśmy przed ich użyciem, to takich tablic nie trzeba w programie zadeklarować. Nawet jeśli tablice zostały wcześniej zadeklarowane instrukcją **DIM** i nadano im inne wymiary niż wymiary tablicy zapisanej na taśmie, to ładowanie unieważnia deklarację. Załadowana tablica będzie miała oczywiście takie wymiary i wartości, jakie nadano jej w programie, który zapamiętał ją na taśmie. Użycie w miejscu nazwy łańcucha pustego " " powoduje próbę wczytania z taśmy magnetofonowej pierwszego napotkanego zbioru typu **DATA**.

Oprócz programów i danych można zapisywać na taśmie jeszcze jeden rodzaj zbiorów: kopie (mapy) wybranego fragmentu pamięci komputera. Realizujemy to instrukcją: **SAVE "nazwa" CODE aaaa, nnnn**, gdzie **aaaa** oznacza adres (numer) pierwszej komórki pamięci, którą należy zapisać, **nnnn** jest zaś liczbą komórek do zapisania. Instrukcja:

SAVE "Blok 1" CODE 30000, 1000

spowoduje zapisanie na taśmie komórek nr 30000 do 30999. Ponownie załadować zapisany obszar można przez **LOAD "nazwa" CODE**. Obszar ładowany jest w to samo miejsce pamięci, z którego został wysłany na taśmę. Gdy chcemy umieścić blok w innym miejscu, wykonamy **LOAD "nazwa" CODE aaaa**, gdzie **aaaa** — adres pierwszej komórki obszaru, pod który blok ma zostać załadowany do pamięci. Gdy zamierzamy wczytać tylko **nnnn** pierwszych komórek zbioru, posłużymy się formą **LOAD "nazwa" CODE aaaa, nnnn**.

Z uwagi na możliwość wystąpienia wad taśmy, zakłóceń w sieci elektrycznej, chwilowych nieregularności przesuwu taśmy itp. niekiedy w zapisanym zbiorze mogą być błędy. Po zapisie warto więc przewinąć taśmę i zweryfikować poprawność zapisu instrukcją **VERIFY**. Instrukcja ta działa podobnie do **LOAD**, jednak odczytywane dane nie są zapisywane do pamięci, lecz jedynie porównywane z jej zawartością. Jeśli wartości odczytane z taśmy pokrywają się z zawartymi w pamięci, dane zapisane zostały poprawnie. W przeciwnym razie zasygnalizowany zostanie błąd, co pozwoli powtórzyć zapis.
Weryfikując poprawność zapisu programu, wystarczy wprowadzić samo **VERIFY "nazwa"** lub **VERIFY " "**. W przypadku danych trzeba zastosować pełną formułę, włącznie z podaniem nazwy tablicy: **LOAD "Tab.A" DATA A()**. Przy weryfikacji bloków pamięci wystarczy **VERIFY "nazwa" CODE**.

Na tym kończymy nasze repetytorium z języka BASIC w ZX Spectrum.

Niektóre rozkazy, jak np. **FORMAT**, **MOVE**, **ERASE**, pominęliśmy. Ich wykorzystanie jest możliwe dopiero po dołączeniu **INTERFACE 1** i zwykle jeszcze stacji mikrodrajwów. Innymi spośród pominiętych rozkazów i funkcji, zwłaszcza związanymi z ingerencją w wewnętrzne życie komputera, zajmiemy się szczegółowiej w dalszej części książki.

Wewnętrzne życie ZX Spectrum

Język BASIC w ZX Spectrum jest elastyczny i wygodny w użyciu. Często jednak jego możliwości nie wystarczają. Programy w języku BASIC są nieefektywne, działają wolno i zajmują znaczny obszar pamięci. Niektórych przedsięwzięć w tym języku zrealizować po prostu nie można. Wtedy sięga się zwykle po język maszynowy. Programy maszynowe wykonywane są bezpośrednio przez stanowiący serce Spectrum mikroprocesor Z80A. Na szczęście BASIC w Spectrum zawiera furtkę w postaci instrukcji **POKE** i **OUT** oraz funkcji **PEEK**, **USR**, **IN** i **BIN**, pozwalających ingerować bezpośrednio w pamięć maszyny i uruchamiać programy maszynowe.
Nieznajomość języka maszynowego nie przekreśla zupełnie możliwości korzystania z jego walorów. W typowych zastosowaniach korzystamy często z gotowych procedur maszynowych, wykonujących np. krytyczne czasowo zadania graficzne. Najpraktyczniejszym kompromisem jest napisanie właściwego programu w języku BASIC i uzupełnienie go tylko najniezbędniejszymi programami maszynowymi. Przykłady gotowych do użycia procedur maszynowych zawarte są na dalszych stronach tej książki. Aby je wkomponować we własne programy i uruchomić bez "rozkładania na łopatki" systemu, potrzebne będzie minimum wiedzy o wewnętrznej organizacji Spectrum, postaci programów i danych. Wiedza ta przyda nam się też przy programowaniu w języku BASIC, przyczyniając się np. do efektywniejszego pomieszczenia informacji w pamięci lub zabezpieczenia naszego programu przed osobami niepowołanymi.

Z wizytą w tajemniczym "PEEKistanie"

Pamięć ZX Spectrum zorganizowana jest w postaci zbioru identycznych komórek, ponumerowanych od Ø do 32767 (wersja 16 KB) lub do 65535 (wersja 48 KB). W każdej z tych komórek może być zapisana niewielka liczba całkowita — od Ø do 255. Pierwszych 16384 komórek tworzy pamięć ROM (READ ONLY MEMORY). Ich zawartość określona została już na etapie

produkcji. Można z nich informację odczytywać, nie można jednak ich zawartości zmodyfikować.
Pozostałe komórki to pamięć RAM (RANDOM ACCESS MEMORY), pozwalająca nie tylko na odczyt, ale i na zapis informacji. Zapisana w komórce informacja przechowywana jest aż do momentu wprowadzenia na jej miejsce nowej informacji lub wyłączenia zasilania.
Zapis i odczyt zawartości pojedynczej komórki pamięci zrealizować można instrukcją **POKE** i uzupełniającą ją funkcją **PEEK**. Instrukcja **POKE** potrzebuje dwóch argumentów. Pierwszym jest adres (numer komórki pamięci), drugim — nowa zawartość komórki.
POKE 32760, 45 wpisuje do komórki nr 32760 liczbę 45.
Funkcja **PEEK** odczytuje zawartość komórki pamięci o wskazanym adresie. **PRINT PEEK 1567** informuje nas, jaką liczbę zawiera komórka nr 1567. Wyczyśćmy pamięć zleceniem **NEW**, po czym wprowadźmy i uruchommy poniższy program:

Program 21

```
10 PRINT PEEK  297, PEEK 30000
20 POKE 297,124: POKE 30000, 124
30 PRINT PEEK  297, PEEK 30000
```

W wyniku wykonania instrukcji 10 wyświetlone zostaną liczby: 79 i 0, wskutek instrukcji 30: 79 i 124. Zawartości komórki 297, należącej do obszaru ROM, zmienić nie zdołaliśmy. W komórce **RAM** o adresie 30000 zapisaliśmy natomiast nową wartość.
Dlaczego wartość liczby mieszczącej się w komórce ogranicza się do 255? We wnętrzu komputera niepodzielnie króluje zapis dwójkowy (binarny). Wszystkie informacje zapisywane są za pomocą kombinacji cyfr dwójkowych: bitów. Każdy bit może znaleźć się zaledwie w dwóch stanach, umownie oznaczonych 0 i 1. Stanom tym odpowiadają w maszynie określone zjawiska elektryczne, np. obecność lub brak ładunku elektrycznego lub przepływu prądu.
Ze względów organizacyjnych bity zgrupowane są w większe jednostki, liczące po 8 bitów i zwane bajtami (ang. byte). W większości przypadków składające się na bajt bity traktowane są jako niepodzielna całość i jako takie uczestniczą w różnych operacjach.
Pojedynczy bit może przedstawiać dwie różne informacje, np. liczby 0 i 1. Dwa bity to już 4 kombinacje — każdy bit może niezależnie znajdować się w jednym z dwóch stanów: 00, 01, 10, 11. Za pomocą dwóch bitów przedstawić już można np. liczby od 0 do 3. Z dodaniem każdego kolejnego bitu liczba możliwych kombinacji podwaja się. Liczbę niepowtarzalnych kombinacji (układów) n bitów wyrazić można wzorem: 2^n. 2^8 to 256 kombinacji, co pozwala na zapis 256 różnych liczb: od 0 do 256. Nadszedł czas

oswajania się z zapisem dwójkowym. Najpierw przypomnijmy sobie, co w systemie dziesiętnym oznacza taki ciąg cyfr: 3181. Oto jego rozwinięcie:

$$3*1\emptyset\emptyset\emptyset + 1*1\emptyset\emptyset + 8*1\emptyset + 1*1$$

lub inaczej:

$$3*10^3 + 1*1\emptyset^2 + 8*1\emptyset^1 + 1*1\emptyset^\emptyset$$

$1\emptyset^\emptyset$ to oczywiście 1. Takie są zasady zapisu liczb w systemie dziesiętnym (w systemie z podstawą 1Ø). Zauważmy, że "ważność" (waga) każdej cyfry wzrasta dziesięciokrotnie (o podstawę systemu!) przy przesuwaniu się w lewo od pozycji do pozycji. Pierwsza i trzecia od prawej cyfra to jedynki. Cyfra trzecia jest jednak 1ØØ razy istotniejsza od pierwszej, określając nie liczbę jednostek, lecz setek.
W systemie binarnym rządzą podobne reguły. Jedyna różnica tkwi w podstawie systemu, którą jest nie 1Ø, lecz 2. Waga cyfry rośnie więc przy przesuwaniu się w lewo tylko dwukrotnie. Dopuszczalne cyfry to Ø i 1.
Poniższy ciąg cyfr przedstawia liczbę w zapisie dwójkowym:

$$\emptyset\emptyset1\emptyset11\emptyset1_2$$

Indeks "2" jest tylko wskaźnikiem, w jakim systemie przedstawiono liczbę. Liczba jest ośmiobitowa. Kolejne bity numerowane są od prawej do lewej. Skrajny prawy bit ma numer Ø i nazywany jest bitem najmniej znaczącym lub najmłodszym. Bit skrajny lewy to bit najbardziej znaczący lub najstarszy.
Ustalmy, jak przedstawiona liczba wygląda w systemie dziesiętnym:

$$\begin{aligned} &\emptyset*2^7+\emptyset*2^6+1*2^5+\emptyset*2^4+1*2^3+1*2^2+\emptyset*2^1+1*2^\emptyset \\ &\emptyset + \emptyset + 32 + \emptyset + 8 + 4 + \emptyset + 1 = 45_{1\emptyset} \end{aligned}$$

Zapisy: $\emptyset\emptyset1\emptyset11\emptyset1_2$ i $45_{1\emptyset}$ przedstawiają więc tę samą liczbę w różnych systemach.
Chociaż mikrokomputer operuje wyłącznie na liczbach dwójkowych, jednak w kontakcie z operatorem Spectrum używa systemu dziesiętnego. W tym systemie interpretowane są np. wartości podawane z klawiatury. Ciąg: ØØ1Ø11Ø1, wprowadzony w instrukcji **INPUT**, odczytany zostałby jako sto jeden tysięcy sto jeden. Czy posługując się liczbami dwójkowymi skazani będziemy na ich "ręczne" tłumaczenie na system dwójkowy? Na szczęście nie, gdyż Spectrum posiada funkcję **BIN**. Jej argumentem jest ciąg złożony z 1 do 16 cyfr dwójkowych, który funkcja **BIN** przetwarza na postać wewnętrzną. Oto przykład:

PRINT BIN 1Ø11Ø1

Instrukcja ta wyświetli, zgodnie z oczekiwaniami, liczbę 45. Operacje arytmetyczne na liczbach dwójkowych odbywają się podobnie jak na dziesiętnych. Przeniesienie następuje oczywiście już wtedy, gdy na danej pozycji wystąpi liczba większa nie od 9, lecz od 1:

$$\begin{array}{r} ØØ\ 1ØØ11Ø_2 \\ +\ Ø\ 1Ø\ 1Ø11\ 1_2 \\ \hline Ø1111Ø1_2 = 125_{1Ø} \end{array}$$

Uzyskany wynik łatwo sprawdzić używając funkcji **BIN**:

PRINT BIN 1ØØ11Ø +BIN 1Ø1Ø111

Pozornie dziwny wydaje się fakt, że Spectrum radzi sobie z liczbami bardzo dużymi (np. $1Ø^{38}$), mimo że elementarna komórka pamięci mieści zaledwie 255. W razie potrzeby jednak można łączyć kilka komórek. Połączenie dwóch jednobajtowych komórek daje nam 16 bitów, pozwalających zapisać liczby od Ø do 65535. Właśnie tyle, ile komórek w pamięci Spectrum... Widzimy, że system binarny wszędzie odciska swoje piętno. We wnętrzu mikroprocesora Z80 do przedstawiania adresów wykorzystuje się dwa bajty. Stąd liczba możliwych adresów. Aby unikać dużych liczb, liczbę $2^{1Ø} = 1Ø24$ nazywa się **kilobajtem**. Dla odróżnienia od właściwego **kilo**, oznaczającego dokładnie 1ØØØ, symbolem informatycznego **Kilo** jest nie małe, lecz duże **K**. Zamiast pisać: 65535 bajtów, wystarczy zanotować: 64 KB.

Język BASIC wykorzystuje do zapisu pojedynczej liczby 5 bajtów. 4 z nich przechowują tzw. mantysę ("cyfry znaczące"), piąty — wykładnik. Spectrum wykorzystuje bowiem zapis wykładniczy, jak np. w systemie dziesiętnym $1.73 \cdot 1Ø^{19}$.

Pięć bajtów dla jednej liczby zapewnia dziewięciocyfrową dokładność. W wielu programach trzeba zapamiętać dużą ilość małych liczb całkowitych. Poświęcanie na każdą z nich aż 5 bajtów jest marnotrawstwem. Jeśli liczby mieszczą się w jednym bajcie, można zapamiętywać je w wybranym obszarze pamięci pojedynczą instrukcją **POKE**. Gdy mamy do czynienia z liczbami większymi, ale nie przewyższającymi 65535, trzeba użyć dwóch bajtów. Jak rozbić liczbę dziesiętną na dwa bajty, odpowiadające jej dwójkowemu przedstawieniu?

Zauważmy, że starszy (bardziej znaczący) bajt informuje o tym, ile pełnych dwieściepięćdziesiątekszóstek mieści się w danej liczbie. Młodszy (mniej znaczący) bajt określa liczbę jednostek nie mieszczących się w całkowitej wielokrotności 256. Sprawdźmy:

PRINT BIN 1Ø1ØØØ1ØØ11

Rezultatem będzie 1299. Starszy bajt ma postać: ØØØØØ1Ø1 (dla czytelności dodano nie znaczące zera), młodszy — ØØØ1ØØ11. **BIN 1Ø1 = 5, BIN 1ØØ11** = 19. Istotnie, $5 \cdot 256 + 19 = 1299$.

Aby uzyskać starszy bajt, wystarczy po prostu podzielić liczbę przez 256, a z uzyskanego ilorazu wziąć część całkowitą. Młodszy bajt otrzymamy, odejmując od liczby wartość starszego bajtu pomnożoną przez 256. Dla ułatwienia sobie życia w przyszłości proponuję zdefiniować prościutki podprogram, zapisujący w dwóch kolejnych komórkach pamięci dwubajto-

wą liczbę całkowitą. Z podprogramu tego będziemy w przyszłości często korzystać. Nazwiemy go **DPOKE** — od istniejącej w niektórych komputerach instrukcji pozwalającej zapisywać do pamięci dwubajtowe liczby:

Program 22

```
9890 REM  DPOKE(adr, db)
9892 POKE adr, db-256*(INT (db/256))
9894 POKE adr+1, INT (db/256)
9896 RETURN
```

Instrukcję **REM** można oczywiście pominąć. Podprogram potrzebuje dwóch parametrów: **adr** i **db**; **adr** to adres młodszego bajtu. Przyjmijmy, że młodszy bajt umieszczany będzie w komórce o niższym adresie. Taką samą konwencję respektuje mikroprocesor. Drugi parametr, **db**, to liczba całkowita od Ø do 65535, którą chcemy zapisać w pamięci. Dla odczytania dwubajtowej liczby z pamięci zdefiniujemy odpowiednią funkcję **FN D**:

```
9900 DEF FN D(a)= PEEK a + 256*PEEK (a+1)
```

Jej argumentem będzie adres młodszego bajtu odczytywanej liczby. Mamy już pierwsze narzędzia, ułatwiające poruszanie się po niebezpiecznej dla nowicjusza krainie, najeżonej słowami **PEEK** i **POKE**. Podobne narzędzia, w postaci podprogramów i funkcji, tworzyć będziemy po drodze jeszcze nie raz. Numeracja wierszy i nazewnictwo zmiennych w przytaczanych w tej książce podprogramach i definicjach funkcji, nazwy samych funkcji itp. dobrano tak, by nie kolidowały ze sobą. Jednym słowem, wszystkie te funkcje i podprogramy mogą znajdować się równocześnie w pamięci Spectrum. W praktycznych zastosowaniach można wybierać tylko niektóre z nich — aktualnie potrzebne. Praktyczne może więc okazać się zapisanie tych podprogramów i definicji na taśmie jako oddzielnych zbiorów, a następnie dołączanie ich do programu z takiej biblioteczki instrukcją **MERGE**.
Może się zdarzyć, że zechcemy ustalić, jaka jest wartość np. bitu nr 5 w zadanej liczbie dwójkowej. Do testowania n-tego bitu w danej liczbie zdefiniujemy funkcję **FN T**:

```
9902 DEF FN T(W,N)= INT(W/2↑N)<>2*INT(W/2↑(N+1))
```

Pierwszy argument (**W**) to badana liczba, drugi (**N**) — numer testowanego bitu. Przeprowadźmy ze zdefiniowaną funkcją eksperyment:

Program 23

```
10 POKE 30000, BIN 10011101
20 FOR K=7 TO 0 STEP -1
```

```
30   PRINT FN T(PEEK 30000, K);
40 NEXT K
```

Po uruchomieniu programu na ekranie pojawi się ciąg: **10011101**, odpowiadający zapisanej uprzednio w pamięci liczbie. Pętla odlicza wstecz, co wynika z faktu, że bity numerowane są od prawej do lewej, znaki zaś wyświetlane są na ekranie od lewej do prawej. Podobny program możemy zastosować np. do tłumaczenia na postać dwójkową liczb całkowitych nie większych od 65535:

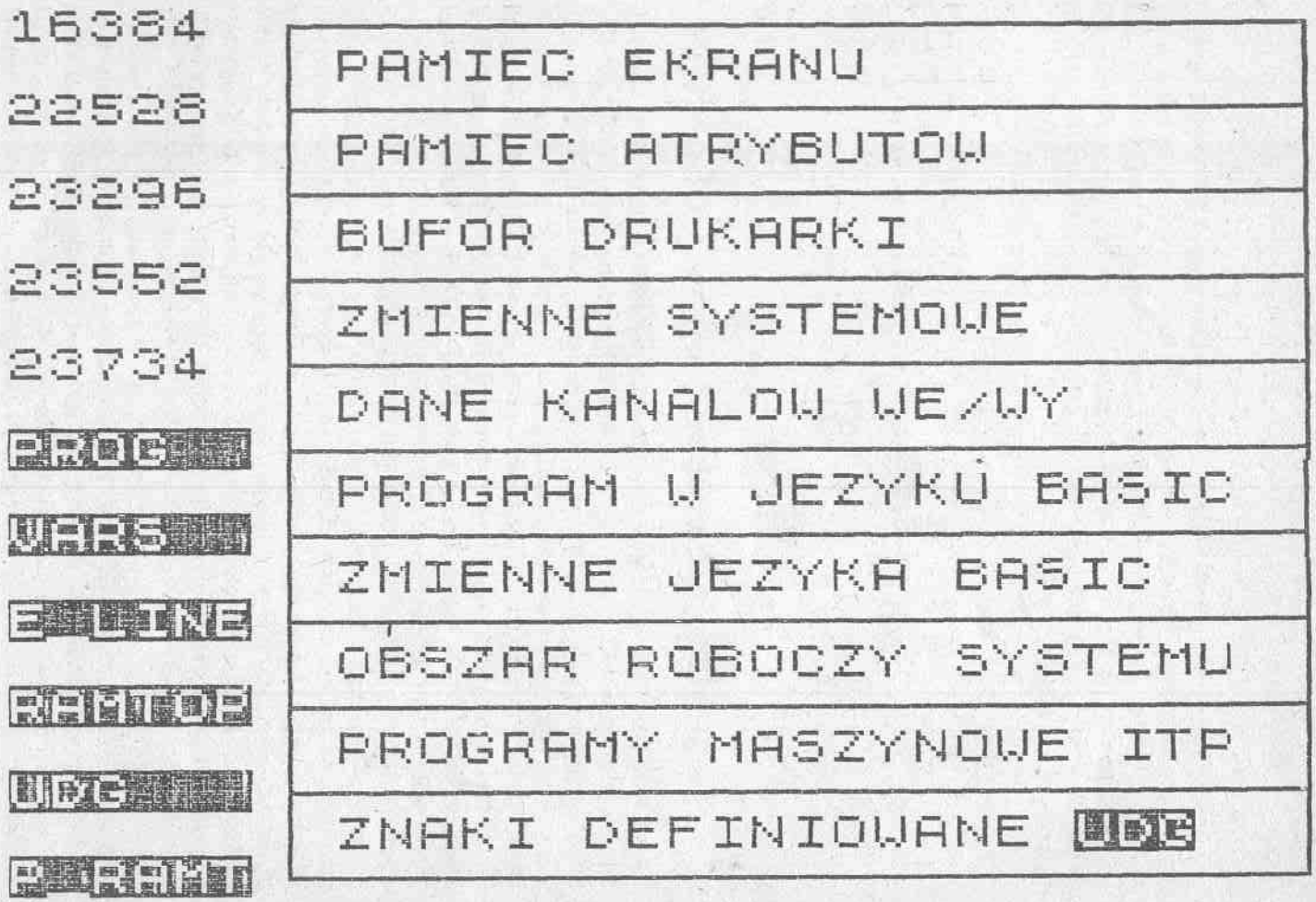
Program 24

```
10 INPUT "Podaj liczbe dziesietna:"; X
20 PRINT "Dwojkowo "; X; "= ";
30 FOR K= 15 TO 0 STEP -1
40   PRINT FN T(X, K);
50 NEXT K: PRINT
```

Organizacja pamięci RAM w ZX Spectrum

Zanim zaczniemy swobodnie buszować w pamięci RAM, poznajmy najpierw zasady jej organizacji. Pamięć ta przechowuje przecież zarówno informację o aktualnej zawartości ekranu, jak i program w języku BASIC wraz ze wszystkimi jego zmiennymi oraz mnóstwo zmiennych specjalnych, jedno- lub dwubajtowych, zwanych **zmiennymi systemowymi**. Zmienne systemowe nie są dostępne bezpośrednio — nie można np. wywołać ich przez nazwę z języka BASIC. Nie jest to jednak zazwyczaj potrzebne. Komórki pamięci, zawierające zmienne systemowe, przechowują informacje do użytku wewnętrznego mikrokomputera. Umiejętnie manipulując zmiennymi systemowymi można uzyskać efekty normalnie niemożliwe w języku BASIC. Nie przemyślana zmiana wartości którejś ze zmiennych systemowych, podobnie jak nieumiejętna ingerencja w wewnętrzną postać programu, zakończyć się może jednak komunikatem o błędzie lub wypadnięciem systemu.

Rysunek 4 przedstawia schemat organizacji pamięci RAM w ZX Spectrum. Pierwszych 6912 bajtów, od adresu 16384 do 23295 włącznie, zajmuje pamięć ekranu. Wkrótce zajmiemy się nią szczegółowo. Następnych 256 bajtów wykorzystywanych jest tylko przy współpracy ze standardową drukarką ZX Printer lub GP50S. Nie korzystając z drukarki możemy tym obszarem dysponować swobodnie, bez ryzyka. Od adresu 23552 do 23733 rozciąga się obszar zmiennych systemowych. Tu wskazana jest najwyższa ostrożność. Dotyczy to także następnych 20 bajtów, mieszczących informację o tzw. kanałach, czyli sposobie wprowadzania i wyprowadzania informacji. Przy dołączonym INTERFACE 1 i korzystaniu z mikrodrajwów lub innych



Rys. 4 Organizacja pamięci RAM w ZX Spectrum

kanałów wejścia—wyjścia obszar ten może się powiększyć. Zawarty pod adresem 23754 bajt wartości 128 (**BIN 10000000**) pełni rolę słupa granicznego między obszarem systemowym a pamięcią programu w języku BASIC, która zaczyna się od komórki 23755. Wewnętrzną budową programu w języku BASIC zajmiemy się później.
Bezpośrednio za ostatnią linią programu znajduje się obszar dla zmiennych języka BASIC, zakończony bajtem wartości 128. Cały obszar zawarty za zmiennymi stanowi roboczą pamięć mikrokomputera, wykorzystywaną np. przy wprowadzaniu danych i programu, wykonywaniu instrukcji **MERGE**, poprawianiu instrukcji itp. Tu przechowywane są też m.in. adresy powrotu, złożone przez instrukcje **GO SUB** w chwili wywołania podprogramu. Umowną granicą tego obszaru jest bajt o adresie zawartym w dwubajtowej zmiennej systemowej **RAMTOP**, przechowywanej pod adresem **23730**. Wskazany bajt jest ostatnim bajtem obszaru dostępnego dla mikrokomputera pracującego w języku BASIC. To właśnie zmienna **RAMTOP** otrzymuje nową wartość przy wykonywaniu instrukcji **CLEAR** z argumentem. Wyłączmy komputer i włączmy go ponownie (jeśli mamy przycisk RESET, użyjmy lepiej jego). Wprowadźmy z klawiatury definicję funkcji **FN D**. Wykonajmy zlecenie:

PRINT FN D (23730)

Odpowiedzią będzie 65367 w wersji 48 KB i 32599 w wersji 16 KB. Wykonajmy zlecenie:

CLEAR 30000 : PRINT FN D(23730)

Tym razem odpowiedzią będzie właśnie 30000. Bajty powyżej 30000 są chronione. Nie niszczy ich nawet zlecenie **NEW**. Pamiętajmy, że obniżenie wartości **RAMTOP** ogranicza miejsce dla programu i danych w języku BASIC. Wykonajmy:

NEW : CLEAR 23880

Spróbujmy teraz wprowadzić z klawiatury jakikolwiek program. Już po kilku instrukcjach przekonamy się, że komputer zasygnalizuje błąd **G : No room for line** (brak miejsca w pamięci dla wiersza programu). Ostatnich 168 komórek pamięci (wersja 48 KB — od 65368, 16 KB — od 32600) mieści normalnie wzorce tzw. znaków definiowanych przez użytkownika (ang. USER DEFINED GRAPHICS — UDG). Jeśli nie stosujemy znaków definiowanych, możemy wykorzystać także i ten obszar.
Skąd mikrokomputer wie, gdzie znajdują się granice poszczególnych obszarów? Adresy granicznych komórek przechowywane są w dwubajtowych zmiennych systemowych. Zmienna **PROG** (adres: 23635) zawiera adres pierwszego bajtu programu w języku BASIC (niekiedy może on zaczynać się w innym miejscu niż komórka 23755). Zmienna **VARS** (adres: 23627) przechowuje adres pierwszego bajtu obszaru zmiennych języka BASIC

(pierwszego bajtu za ostatnim znakiem programu). Adres wzorców UDG umieszczony jest w zmiennej systemowej **UDG** (adres: 23675). Istnieje jeszcze wiele wskaźników, chwilowo są one jednak mniej istotne. Będziemy je omawiać w razie potrzeby.

Pamięć ekranu

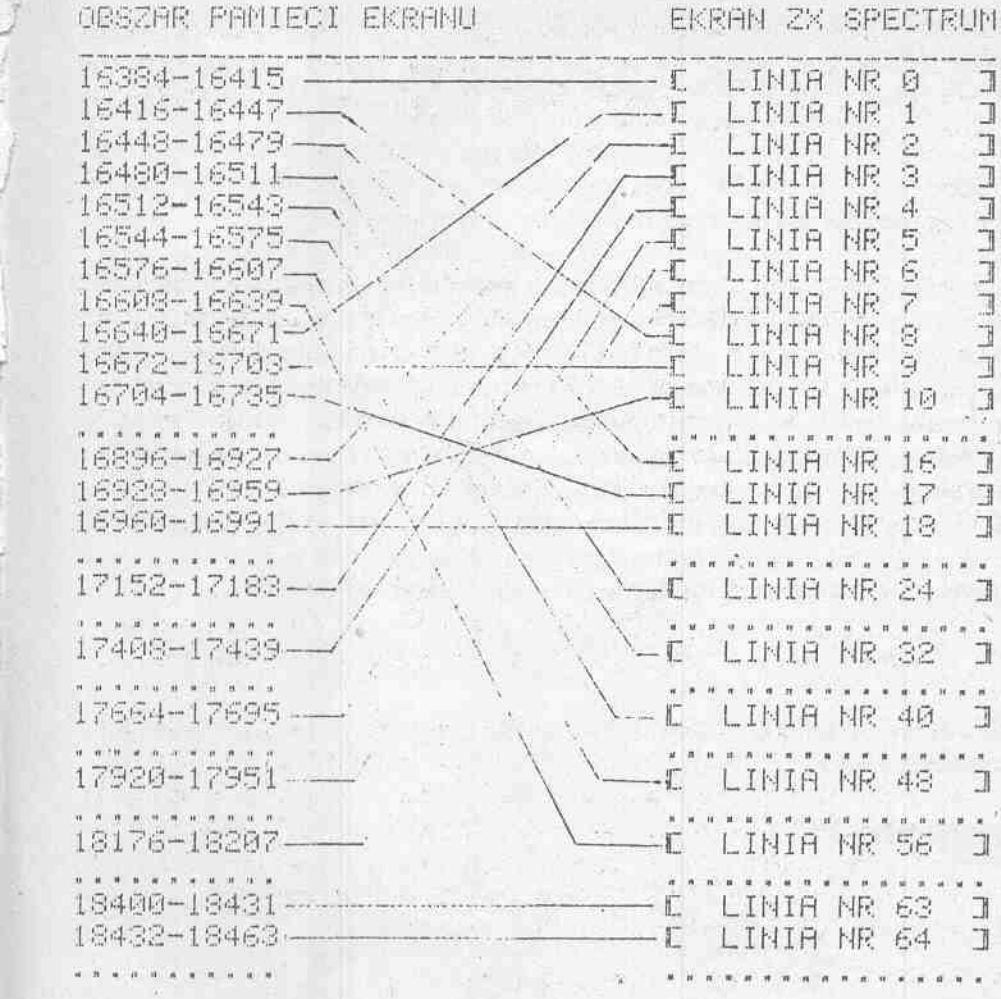
Najbardziej interesującym obszarem pamięci RAM jest dla nas chwilowo pamięć ekranu. Jeśli Czytelnik miał nadzieję, że bity i bajty są abstrakcją rezydującą wyłącznie w hermetycznym wnętrzu maszyny i nigdy nie grozi mu bezpośrednia z nimi konfrontacja, to zaraz wyprowadzimy go z błędu.
Wiemy już, że ekran Spectrum tworzy siatka zawierająca 192 linie po 256 punktów, zwanych też pikselami (ang. pixel). Pamiętajmy, że punkt może być wygaszony (barwa tła) lub zapalony (barwa atramentu). Za stan każdego z 49152 punktów ekranu odpowiedzialny jest jeden, ściśle określony bit pamięci.
Promień elektronów kreśli obraz na ekranie kineskopu linia po linii, od lewej do prawej i z góry na dół. Moglibyśmy oczekiwać, że odpowiedzialne za ekran bity uporządkowane są w sposób odpowiadający kolejności kreślenia linii.
256 bitów składających się na jedną linię zawiera się w 256/8= 32 bajtach. Każdy bajt mieści informację o stanie 8 kolejnych punktów. Bit nr 7 (najstarszy) kontroluje najbardziej lewy punkt itd. Tworzące linię bajty ułożone są w pamięci po kolei: bajt lewego końca linii jako pierwszy, bajt prawego końca jako ostatni. Pierwsza linia ekranu zapamiętana jest w pierwszych 32 bajtach pamięci ekranu, od adresu 16384. Wykasujmy ekran zleceniem **CLS** i wpiszmy jakąś zawartość do pierwszej komórki tego obszaru:

POKE 16384, BIN 11111111

W lewym górnym rogu ekranu ukaże się pozioma kreseczka. Po zleceniu:

POKE 16415, BIN 1111ØØ11

w prawym górnym rogu pojawi się także układ punktów, odpowiadający kompozycji bitów zadanej w funkcji **BIN**.
Logiczne byłoby, gdyby przedstawiające poszczególne linie grupy 32 bajtów następowały w pamięci kolejno po sobie. Niestety, nie jest tak z powodów, które omówimy później. Na razie zapamiętajmy, że pamięć ekranu dzieli się na trzy identyczne bloki, odpowiedzialne za trzy szerokie, poziome pasy po 64 linie każdy: górny, środkowy i dolny. Każdy blok zajmuje dokładnie 2 KB (32×64 bajty). Ponumerujmy linie w obrębie pasa od Ø do 63, poczynając od góry. Pierwsze 32 bajty odpowiadają za linię nr Ø. Następne 32 bajty określają linię nr 8. Kolejne 32 bajty — linię 16, 24 itd. aż do linii 56.
Kolejnych 256 bajtów określa w podobny sposób wygląd linii 2, 9, 17... 57. Ostatnich 256 bajtów w bloku odpowiada za linie 7, 15, 23... 63 (rys. 5).



Rys. 5 Przyporządkowanie obszarów pamięci liniom ekranu w ZX Spectrum

By lepiej uzmysłowić tę pozornie dziwaczną organizację, napiszemy krótki program, zapełniający pamięć ekranu:

Program 25

```
10 CLS
20 FOR A= 16384 TO 22527
30   POKE A, BIN 11111111
40 NEXT A
```

Program wypełnia ekran w takiej kolejności, jaka wynika z rozmieszczenia w pamięci bajtów odpowiedzialnych za poszczególne jego fragmenty. Pamięć zapełniania jest przecież kolejno. Podobny obraz obserwujemy przy

ładowaniu z taśmy magnetofonowej np. nagłówka programu. Wtedy pamięć także wypełniana jest kolejno.
Powód, dla którego przyjęto taką organizację pamięci, wynikał ze sposobu określenia atrybutów poszczególnych pól ekranu. Informacja o kolorach, jaskrawości czy migotaniu każdego z 768 pól zawarta jest w 768 ostatnich bajtach pamięci ekranu — tzw. obszarze atrybutów, od 22528 do 23295. Tutaj organizacja jest już rozsądna: górnemu wierszowi z 32 pól odpowiadają pierwsze 32 bajty, następnemu — kolejne 32 bajty itd. Każdemu blokowi, tworzącemu pas 64 linii, czyli 8 wierszy znakowych, towarzyszy dokładnie 256 bajtów atrybutów. Mamy oto rozwiązanie zagadki: wszystkie bajty, mieszczące informacje o punktach jednego pola, mają identyczny, młodszy bajt adresowy (sprawdźcie na rys. 5), zgodny z młodszym bajtem adresu odpowiadającego im bajtu atrybutów! Istotnie upraszcza to proces pobierania z pamięci informacji o układzie punktów i atrybutach w trakcie wyświetlania. Zajmuje się tym specjalizowany układ scalony zwany **ULA**.
Zdefiniujmy teraz funkcję, która odpowie nam na pytanie, od jakiego adresu zaczyna się grupa bajtów odpowiedzialna za linię ekranu o danym numerze (linie ponumerowaliśmy od Ø do 191, poczynając od góry):

```
9904 DEF FN M(y)=16384+2048*INT (y/64)+256*
(y-8*INT (y/8))+32*(INT (y/8)-8*INT (y/64))
```

Wypróbujmy działanie funkcji, zapisując np. drugi bajt każdego wiersza stałą kompozycją bitów:

Program 26

```
10 FOR i=Ø TO 191
20 POKE FN M(i)+1,BIN 11111111
30 NEXT i
```

Na ekranie pojawi się ciemny pas, który będzie płynnie wydłużał się ku dołowi. Fakt, że do wartości **FN M** dodajemy w instrukcji **2Ø** nie 2, lecz 1, wynika z faktu, że pola w wierszu numerujemy od Ø. Drugi bajt ma więc w rzeczywistości numer 1.
Funkcja standardowa **ATTR** pozwala badać, jakie są atrybuty określonego pola. Nam jednak przydałaby się inna funkcja, wyznaczająca adres bajtu atrybutów związanego z danym polem. Zdefiniujmy ją zatem:

```
9906 DEF FN A(w,k)= 22528+32*W+K
```

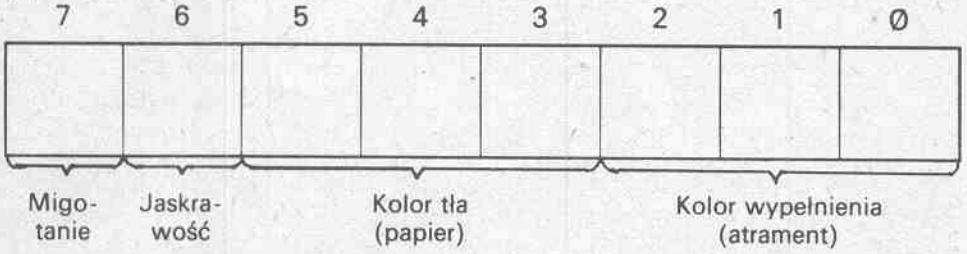
Parametr **W** jest numerem wiersza, **K** — numerem kolumny. Liczba 22528 to oczywiście adres początkowy obszaru atrybutów.
Pora na eksperyment: wykonajmy w trybie bezpośrednim instrukcje:

PAPER 7: INK Ø: BRIGHT Ø: FLASH Ø: CLS :PRINT "AB"

Jeśli od chwili włączenia komputera nie zmieniliśmy atrybutów, to cztery pierwsze instrukcje nie są potrzebne. Napiszmy teraz:

PRINT ATTR (Ø, Ø)

W rezultacie badania atrybutów pola, w które dopiero co wpisaliśmy literę **"A"**, otrzymujemy wynik 56. Przyjrzyjmy się bliżej budowie bajtu atrybutów (rys. 6):



Rys. 6 Budowa bajtu atrybutów

Dla zakodowania kolorów papieru i atramentu przewidziano po 3 bity. Wystarcza to akurat do zapisania 8 kolorów od Ø ($=ØØØ_2$) do 7 ($=111_2$). Jedynki w bitach jaskrawości i migotania oznaczają włączenie danego atrybutu, zera — wyłączenie. Przypomnijmy sobie program 23. W podobny sposób wyświetlimy teraz na ekranie dwójkową postać bajtu atrybutów pola Ø, Ø:

Program 27

```
10 CLS : PRINT "AB"
20 FOR K=7 TO Ø STEP -1
30   PRINT FN T(ATTR (Ø,Ø), K);
40 NEXT K
```

Uruchomiony program potwierdza nasze przypuszczenia: na ekranie pojawia się ciąg ØØ111ØØØ. Jaskrawość i migotanie wyłączone, pole barwy tła zawiera $111_2 = 7_{10}$ (**PAPER 7** — biały), pole barwy atramentu zawiera ØØØ (**INK Ø** — czarny). Wprowadźmy teraz zlecenie:

BRIGHT 1 : PAPER 6 : INK 2

i uruchommy program 27 ponownie. Wynikiem będzie Ø111ØØ1Ø. $6_{10}=11Ø_2$, $2_{10}=Ø1Ø_2$.

Przy użyciu normalnych środków atrybuty danego pola można w języku BASIC ustalać tylko przed zapisaniem w nich nowej zawartości. Funkcja **FN A**, znajomość zapisu dwójkowego i organizacji pamięci pozwolą nam zmienić ten stan. Przywróćmy poprzednie atrybuty zleceniem:

BRIGHT Ø : PAPER 7 : INK Ø

i napiszmy prosty program zapełniający ekran powtarzającą się grupą znaków:

Program 28a

```
10 CLS
20 FOR I=0 TO 21
30   FOR K=1 TO 4
40     PRINT "01234567";
50   NEXT K
60 NEXT I
```

Pragniemy teraz zmienić atrybuty wszystkich znaków w okienku utworzonym z pól 8—23 wierszy od 4 do 15. Żądamy, aby znaki w tym prostokącie migotały, kolor tła był błękitny (=5), barwa zaś znaku — granatowa (=1). $5_{10}=101_2$, $1_{10}=001_2$. Możemy już zestawić dla okna nowy bajt atrybutów: 10101001_2. Teraz dopiszemy jeszcze dwie pętle, które zmodyfikują atrybuty w założonym prostokącie:

Program 28b

```
 70 FOR I=4 TO 15
 80   FOR K=8 TO 23
 90     POKE FN A(I,K),BIN 10101001
100   NEXT K
110 NEXT I
```

Bardziej skomplikowana sytuacja powstaje, gdy chcemy ustalić wartość tylko jednego z atrybutów w wybranym polu. Funkcja **ATTR** dostarcza tylko jedną wartość, stanowiąca kompozycję wszystkich czterech atrybutów. Ustalenie wartości obu atrybutów jednobitowych jest proste. Można posłużyć się przecież funkcją **FN T**, testując wybrane bity. By ustalić np. stan atrybutu jaskrawości (BRIGHT) w polu nr 11 wiersza nr 3, napiszemy:

PRINT FN T(ATTR(3, 11), 6)

Za jaskrawość odpowiada właśnie bit 6 (rys. 6).

Barwę tła i wypełnienia opisują grupy trzybitowe. Dla wyodrębnienia tych kolorów zdefiniujemy funkcję **FN I** dla atramentu (INK) oraz **PN P** (PAPER) dla papieru. Argumentami obydwu funkcji są współrzędne danego pola — jak w **ATTR**.

```
9900 DEF FN I(w,k)=ATTR (w,k)-8*INT (ATTR (w,k)/8)
9910 DEF FN P(w,k)=INT ((ATTR (w,k)-64*INT (ATTR (w,k)/64))/8)
```

Wyobraźmy sobie następującą sytuację. We wszystkich polach ekranu, w których tło jest błękitne (INK 5), zamierzamy podnieść jaskrawość, wpisując w nie równocześnie literę "X". Funkcja **FN P** pozwoli komputerowi "widzieć" barwy pól ekranu. Na początku wypełnijmy ekran spacjami o przypadkowo wybranym kolorze tła:

Program 29a

```
10 CLS
20 FOR I=1 TO 704
30   PRINT PAPER 8*RND; " ";
40 NEXT I
```

Teraz możemy przystąpić do analizy atrybutów we wszystkich polach okna użytkownika. W razie napotkania barwy tła równej 5 wpiszemy w dane pole literę "X" zwiększając jaskrawość (**BRIGHT 1**), lecz nie zmieniając tła (PAPER 8):

Program 29b

```
50 FOR W=0 TO 21
60   FOR K=0 TO 31
70     IF FN P(W,K)=5 THEN PRINT AT W,K; BRIGHT 1; PAPER 8;"X";
80   NEXT K
90 NEXT W
```

Potrzeba "widzenia" ekranu występuje często w różnego rodzaju grach komputerowych, gdy podstawowym zbiorem informacji nie są tablice i zmienne proste, lecz właśnie przedstawiająca aktualną sytuację w grze pamięć ekranu.

Jak powstają znaki

Opanowaliśmy zapalanie lub wygaszanie pojedynczych punktów ekranu oraz odczyt i modyfikację atrybutów. Czas więc przyjrzeć się, w jaki sposób w ekran wpisywane są pojedyncze znaki i zbudowane z nich teksty. Najmniejszym wycinkiem ekranu, na który możemy oddziaływać instrukcją **PRINT**, jest pole 8×8 punktów wraz z odpowiadającymi mu atrybutami. Aby w polu o współrzędnych Ø, Ø (lewy górny róg ekranu) pojawiła się litera "A", trzeba wypełnić właściwą treścią osiem odpowiedzialnych za to pole bajtów pamięci ekranu. Skąd komputer wie, jak narysować literę "A"? Otóż komputer w pamięci ROM ma ściągaczkę w postaci tablicy zawierającej wzorcowe zestawy bajtów — po 8 dla każdego z 96 znaków o kodach od 32

(spacja) do 127 (Copyright). Aby wyświetlić dany znak, wystarczy odszukać odpowiadający mu, ośmiobajtowy wzorzec i wpisać go we właściwe komórki pamięci ekranu.
Zestaw znaków zawartych w ROM jest bogaty, lecz nie zawsze wystarcza. W grach potrzebne są przeważnie specjalne, wymyślne symbole graficzne. Przy redagowaniu tekstów w języku polskim przydatne są znaki głosek ą, ę, ś, ń, ć itd. Przewidując podobne sytuacje, zarezerwowano w Spectrum 21 kodów od 144 do 164. Odpowiadające tym kodom znaki może sobie użytkownik zdefiniować we własnym zakresie. Znaki te zwane są w skrócie UDG (co wyjaśniliśmy już na str. 53). Wzorce tych znaków zawarte są nie w pamięci ROM, lecz w zarezerwowanym obszarze pamięci RAM. Umożliwia to programową zmianę wyglądu tych znaków. Aby zdefiniować kształt dowolnego spośród 21 znaków UDG, wystarczy zapełnić 8 bajtów odpowiadającego znakowi wzorca odpowiednią kompozycją bitów.
Znaki graficzne wywołujemy, przełączając kursor w tryb **G**, po czym wciskając jeden z klawiszów "A" do "U". Początkowa zawartość pamięci wzorców UDG odpowiada właśnie tym symbolom.
Wszystkie wzorce zajmują 8 kolejnych bajtów pamięci. Pierwszy odpowiada najwyższej linii pola itd. Skąd jednak wiadomo, pod jakim adresem znajduje się wzorzec danego znaku UDG? Adres obszaru wzorców UDG zawarty jest w dwubajtowej zmiennej systemowej **UDG** pod adresem 23675. Możemy łatwo odczytać go naszą funkcją **FN D**:

PRINT FN D(23675)

Na początku tego obszaru znajduje się 8 bajtów wzorca znaku o kodzie 144 (pierwotnie "A"), potem UDG o kodzie 145 itd.
Adres pierwszego bajtu wzorca danego znaku UDG najprościej ustalić funkcją **USR**. Jeśli argumentem tej funkcji jest łańcuch, złożony z pojedynczych liter "a" do "u" (litery małe i duże traktowane są tu równorzędnie), to wartością funkcji jest właśnie adres wzorca. Aby dowiedzieć się, gdzie zlokalizowany jest wzorzec UDG "c" (kod 146), wystarczy zlecenie:

PRINT USR "c"

Postarajmy się teraz przedefiniować któryś ze znaków UDG, np. "b":

Program 29c

```
10 FOR A=USR "b" TO USR "b"+7
20   READ bajt: POKE A,bajt
30 NEXT A: PRINT CHR$ 145
40 DATA BIN 11111111
50 DATA BIN 10000001
60 DATA BIN 10111101
70 DATA BIN 10100101
```

```
 80 DATA BIN 10100101
 90 DATA BIN 10111101
100 DATA BIN 10000001
110 DATA BIN 11111111
```

Od tej pory, po naciśnięciu w trybie graficznym klawisza "B", na ekranie będą pojawiać się 2 kwadraty, zawarte jeden w drugim. Dzięki funkcji **BIN** zakodowanie zaprojektowanego np. na kartce symbolu nie przedstawia trudności. Nie zawsze musimy wymyślać kształt znaku od podstaw.
Aby zdefiniować np. literę "ą", wystarczy dorobić ogonek do istniejącego już w repertuarze znaków Spectrum "a". Odwzorować układ bitów małej litery na ekranie nie jest wcale prosto. Lepiej dopomóc sobie krótkim programem w języku BASIC. Załóżmy, że interesujący nas znak wyświetlony został w lewym dolnym rogu okna użytkownika. Zadaniem naszego programu będzie powiększyć go ośmiokrotnie i umieścić na środku ekranu. Na początku program zapyta nas o powiększany symbol:

Program 30

```
10 INPUT "Zadany symbol: "; z$
20 CLS : PRINT AT 21,0; z$
30 FOR y=0 TO 7
40   FOR x=0 TO 7
50     IF POINT (x,y)=1 THEN PRINT AT 15-y,12+x;"X";
60   NEXT x
70 NEXT y
```

Pole (Ø, Ø) składa się z punktów o graficznych współrzędnych Ø—7, zarówno dla X, jak i dla Y. Analizujemy więc kolejne punkty tego pola od dołu do góry i od lewej do prawej. W razie wykrycia punktu zapalonego zapisujemy odpowiednie pole w kwadracie utworzonym przez pola 12—19 w wierszach 7—15. Każde pole tego dużego kwadratu odpowiada jednemu punktowi graficznemu w analizowanym kwadracie pola (Ø, Ø). Wyrażenia **15−y**, **12+x** wyznaczają związek między współrzędną punktu a położeniem odpowiadającego mu pola w powiększonym kwadracie. Znak "−" przed y wynika z faktu, że współrzędna Y wzrasta od dołu do góry, numer zaś wiersza — od góry do dołu. Zamiast litery można użyć dowolnego symbolu graficznego, np. pełnego pola (**CHR$ 143**). Nasz program ukazuje tylko punkty wypełnione, nie sygnalizuje natomiast punktów o barwie tła. Najprościej zaradzić temu, wypełniając wstępnie każde pole powiększonego kwadratu np. żółtym tłem. Wystarczy w tym celu dodać instrukcję:

```
45   PRINT AT 15-y,12+x; PAPER 6; " ";
```

Czy nie można zrezygnować z pośrednictwa ołówka i skopiować wzorca wybranego znaku do obszaru UDG bezpośrednio z pamięci ROM? Chodzi w końcu jedynie o przepisanie 8 kolejnych bajtów!
Zgoda. Potrzebna jest nam tylko informacja o adresie odpowiedniego wzorca w pamięci ROM.
Adres zbioru wzorców, zwanego często generatorem znaków, zawarty jest w dwubajtowej zmiennej systemowej **CHARS** (adres: 23606). Zawartość **CHARS** jest mniejsza o 256 od adresu pierwszego wzorca, przedstawiającego spację. Dlaczego?
Znaki o kodach od Ø do 31 nie mają odpowiedników w postaci możliwych do przedstawienia na ekranie symboli. Nie ma więc potrzeby zapamiętywania ich wzorców. Wzorce istnieją tylko dla znaków o kodach od 32 do 127. Aby wyznaczyć położenie wzorca znaku o zadanym kodzie, trzeba ten kod pomnożyć razy 8 (1 znak wymaga ośmiu bajtów), odjąć od niego 32•8=256, a następnie całość dodać do początkowego adresu generatora znaków. Aby uniknąć odejmowania 256 przy każdym znaku, uwzględniono tę wartość już w zawartości komórki **CHARS**. Zdefiniujemy teraz funkcję wyznaczającą adres wzorca zadanego znaku w pamięci:

```
9912 DEF FN W(c$)=8*CODE c$+FN D(23606)*(CODE C$<128)
     +(USR "a"-1152)*(CODE c$>=144)
```

W konstrukcji funkcji zastosowaliśmy znane nam już przełączniki logiczne. Jeśli kod znaku jest mniejszy od 128, jego wzorzec leży w głównym bloku wzorców, wskazywanym przez **CHARS**. Jeśli natomiast kod znaku jest większy od 143, chodzi prawdopodobnie o symbol UDG.Tajemnicza stała 1152 to po prostu 8•144 (144 jest kodem pierwszego ze znaków UDG). Musimy ją odjąć od pomnożonego już przez 8 kodu znaku, aby dla kodu 144 wyznaczony adres pokrył się z wartością **USR "a"** itd. Rezygnując ze znaków graficznych, możemy funkcję **FN W** uprościć:

```
9912 DEF FN W(c$)=8*CODE c$+FN D(23606)
```

Możemy teraz bez trudu skopiować np. literę "a" do pola wzorca UDG "a", a następnie, modyfikując zaledwie jeden bajt, stworzyć "ą" przez dodanie ogonka:

Program 31

```
10 FOR I=0 TO 7
20   POKE USR "a"+I,PEEK (FN W("a")+1)
30 NEXT i
40 POKE USR "a"+7,BIN 00000110
50 PRINT "a";CHR$ 144
```

Przenieść znak na ekran potrafilibyśmy teraz już bez użycia **PRINT**. W niektórych zastosowaniach mankamentem **PRINT** jest możliwość umieszczania znaków tylko w określonych polach, na określonym poziomie. Niekiedy byłoby celowe zapisanie czegoś między wierszami. Przykładem może być chociażby tradycyjny zapis potęgowania, np. 2^3, lub indeksu x_s. Przygotujmy więc podprogram, który zapisuje znak w dowolnej kolumnie ekranu (Ø—31), w zadanym wierszu (Ø—21) oraz przy określonym przesunięciu względem normalnego poziomu w tym wierszu. Podprogram nazwiemy **LOKUJZNAK**:

Program 32

```
9880 REM LOKUJZNAK(WZ, KZ, PZ, Z$)
9882 LET q1=FN W(Z$)
9884 FOR q=Ø TO 7
9886   POKE FN M(8*WZ-PZ+q)+kZ, PEEK (q1+q)
9888 NEXT q
9899 RETURN
```

Parametry oznaczają odpowiednio: **Z$** — znak, który zamierzamy wyprowadzić, **WZ** — wiersz, **KZ** — kolumna, w której znak ma się pojawić, **PZ** — liczba punktów, o jakie znak będzie przesunięty w górę lub w dół względem poziomu wiersza. Jeśli **PZ**=Ø, znaki wyprowadzane są jak przez instrukcję **PRINT AT WZ,KZ**. Nasz podprogram nie modyfikuje oczywiście żadnych atrybutów. Ujemna wartość **PZ** oznacza, że znak znajdzie się poniżej, dodatnie **PZ** — powyżej osi wiersza. Oto przykład zastosowania:

Program 33

```
10 LET WZ=10
20 PRINT AT 10,2;"x";
30 LET Z$="2": LET KZ=3: LET PZ=4
40 GO SUB 9880
50 PRINT AT 10,10;"A";
60 LET Z$="5": LET KZ=11: LET PZ=-5
70 GO SUB 9880
```

Po uruchomieniu programu na ekranie ukażą się symbole: x^2 i A_5. Nasz podprogram możemy z powodzeniem wykorzystywać np. do ukośnego przedstawiania tekstów:

Program 34

```
10 INPUT "Tekst:"; T$
20 LET WZ=5
30 FOR I=1 TO LEN T$
```

```
40   LET WZ=I-1: LET PZ=I: LET Z$=T$(I)
50   GO SUB 9880
60 NEXT I
70 GO TO 10
```

Program wybiera po jednym znaku z podanego łańcucha **T$**, przekazując je kolejno zmiennej **Z$** — parametrowi podprogramu **LOKUJZNAK**. Za każdym razem wzrasta o 1 numer kolumny **KZ** (znaki umieszczane są w kolejnych kolumnach) oraz przesunięcie **PZ**.

Nowy alfabet — na życzenie

Przy bliższym zastanowieniu się dziwny wydaje się fakt, że mimo umieszczenia generatora znaków na stałe w pamięci ROM adres tego generatora pobierany jest przez mikrokomputer każdorazowo z komórki **CHARS**, umieszczonej w pamięci RAM. Wartość zmiennej umieszczonej w pamięci RAM można modyfikować. Spróbujmy na początek delikatnej zmiany: zwiększmy wartość **CHARS** o 3. Młodszy bajt **CHARS** wynosi normalnie Ø, osiągniemy więc cel, wykonując po prostu zlecenie:

POKE 23ØØ6, 3

Widzimy, co zaszło: od tej chwili wszystkie wyprowadzane znaki są obcięte w górnej części. Nic dziwnego: tylko 5 pierwszych bajtów każdego wzorca jest oryginalnych. 3 następne są pierwszymi bajtami wzorca kolejnego znaku. Wykonajmy zlecenie:

POKE 23606, 8

Znaki wyglądają od tej chwili normalnie, lecz zamiast "a" wyświetlane jest "b", w miejscu cyfry "1" pojawia się "2", a zamiast spacji wyprowadzany jest wykrzyknik (**CODE** " "=32, **CODE** "!"=33). Co się stało? Zwiększyliśmy początkowy adres generatora znaków o 8 bajtów: tyle, ile zajmuje wzorzec jednego symbolu. Wzorzec spacji jest pierwszym ze wzorców. Wyświetlając spację komputer sięga więc po pierwszy znak spod adresu wskazywanego przez **CHARS**. Pod tym adresem znajduje się jednak wzorzec wykrzyknika: **CHARS** wynosi o 8 bajtów więcej niż zwykle.
Samego generatora znaków, zapisanego w ROM, przemieścić nie zdołamy. Mimo to oszukaliśmy komputer, podając mu błędny adres.
Możliwość zmiany adresu generatora znaków może być oczywiście wykorzystana bardziej celowo. Wyobraźmy sobie, że w wybranym obszarze pamięci RAM tworzymy nowy generator znaków. Musimy przestrzegać oczywiście zasad jego organizacji: wzorce są ośmiobajtowe, a ich kolejność odpowiada wzrastającym kodom przedstawianych znaków. Kształty znaków możemy jednak zdefiniować zupełnie dowolnie, według własnego uznania. Znaki

mniej potrzebne możemy np. zastąpić symbolami nie występującymi w standardowym zestawie symboli, a niezbędnymi w naszych zastosowaniach. Następnie przełączamy Spectrum na nowy zestaw znaków, ładując zmienną systemową **CHARS** adresem nowego zbioru wzorców, pomniejszonym oczywiście o 256. Od tej pory Spectrum "zapomina" o zaszytym w pamięci ROM generatorze i pisze po ekranie wyłącznie zaprojektowanymi przez nas znakami.

Przeniesienie generatora znaków do pamięci RAM może być szczególnie celowe wtedy, gdy zależy nam na sprawnym operowaniu polskimi znakami alfabetu. Można by sądzić, że najprościej zdefiniować odpowiednie UDG. Prawda. Takie rozwiązanie będzie jednak bardzo niewygodne w praktycznej eksploatacji. Pamiętajmy, że litery "ą", "ę", "ś", "ó" itp. występują dość często. Dla wprowadzenia znaku należącego do zbioru UDG trzeba najpierw uaktywnić tryb graficzny, naciskając równocześnie CAPS SHIFT i "9", potem wcisnąć klawisz odpowiedniego znaku, w końcu wyłączyć tryb graficzny wciskając ponownie CAPS SHIFT i "9". Wszystko to jest uciążliwe, sprzyja błędom i bardzo wybija z rytmu przy wpisywaniu dłuższych tekstów.

Lepszym rozwiązaniem będzie rezygnacja z niektórych, mniej potrzebnych symboli dostępnych na klawiaturze bezpośrednio i zastąpienie ich znakami często używanymi. Przeważnie poświęcić można bez wielkiej szkody symbole "$", "£", nawiasy klamrowe i prostokątne itp. Każdy z tych znaków dostępny był pojedynczym wciśnięciem klawisza.

Umieszczenie generatora znaków w pamięci ROM może być koniecznością także wtedy, gdy liczba dostępnych znaków UDG okaże się niewystarczająca. Sytuacja taka wystąpić może w programowaniu gier, a także programów edukacyjnych, w których potrzeba greckich liter, symboli chemicznych itp. W większości przypadków nie będziemy tworzyć wszystkich znaków od podstaw, lecz skopiujemy wzorce z pamięci ROM, po czym wprowadzimy odpowiednie poprawki.

Lokalizacja nowego generatora wymaga przemyślenia. 96*8=768 bajtów to sporo. Na ogół dobrym miejscem będzie obszar tuż poniżej UDG. Adres początku generatora otrzymamy w takim wypadku w prosty sposób:

PRINT USR "a" - 768

W niektórych sytuacjach wystarczy zadeklarować dwa lub więcej bloków UDG, a następnie przełączać je w razie potrzeby podając nową wartość zmiennej **UDG**. Pamiętajmy, że **UDG** — w odróżnieniu od **CHARS** — zawiera adres pierwszego bajtu obszaru wzorców UDG (nie pomniejszony o 256). Spróbujmy swoich sił, przenosząc generator znaków do RAM, powiedzmy tuż poniżej UDG. Zaczniemy oczywiście od zarezerwowania instrukcją **CLEAR** odpowiedniego obszaru pamięci zabezpieczonego przed przypadkowym zapisaniem przez interpreter języka BASIC. Następnie pętlą **FOR... NEXT** skopiujemy wszystkie bajty wzorców z ROM do RAM. Do dzieła:

Program 35

```
10 CLEAR USR "a"-769
20 LET noweznaki=USR "a"-768
30 LET stareznaki=FN D(23606)+256
40 FOR I=0 TO 767
50   POKE noweznaki+I, PEEK (stareznaki+I)
60 NEXT I
70 LET db=noweznaki-256: LET adr=23606
80 GO SUB 9890: REM DPOKE
90 STOP
```

Argumentem **CLEAR** jest adres nowego generatora znaków pomniejszony o 1. Argument **CLEAR** wskazuje na ostatni bajt dostępny dla języka BASIC. Bajt ten może zostać przypadkowo zmodyfikowany. Musimy więc wskazać w instrukcji **CLEAR** bajt poprzedzający pierwszy bajt generatora znaków. Po skopiowaniu wzorców procedura **DPOKE** przełączy komputer na nowy zestaw wzorców, zapisując nową zawartość **CHARS**. Z początku nie zaobserwujemy żadnych zmian: nowy i stary generator są przecież identyczne.

Przypuśćmy, że zamierzamy wprowadzić polskie znaki głosek. Wszystkie one wywodzą się od liter już w Spectrum znanych. W pierwszym rzędzie skopiujemy więc wyjściową postać znaków. Znowu posłużmy się funkcją **FN W**:

Program 36

```
100 INPUT "Znak do zastapienia: "; a$
110 IF LEN a$=0 THEN STOP
120 INPUT "Nowy ksztalt znaku: "; b$
130 FOR I=0 TO 7
140   POKE FN W(a$)+I, PEEK (FN W(b$)+I)
150 NEXT I
160 GO TO 100
```

Program rozpoczyna pracę pytaniem o znak do zastąpienia, czyli o znak mniej potrzebny, np. "%", któremu przypiszemy postać "ą", "ę" itp. Podając w odpowiedzi łańcuch pusty (wystarczy od razu wcisnąć ENTER), spowodujemy zatrzymanie pracy programu. W przeciwnym razie nastąpi pytanie o nową postać znaku, który modyfikujemy. Chcąc otrzymać "ą", podamy początkowo "a". Program przepisze 8 bajtów wzorca, po czym zapyta o kolejny znak do modyfikacji. Możemy teraz wprowadzić ostatnie poprawki. Dodając literze "a", skopiowanej np. w miejsce "%", dwa punkty u dołu, otrzymamy wymarzone "ą". Wystarczy tu zlecenie:

POKE FN ("%")+7, BIN ØØØØØ11Ø

Przy wypisywaniu zlecenia na klawiaturze spotka nas zaskoczenie: wciskając "%" spowodujemy ukazanie się znaku "a". Nie sięgajmy po DELETE: sami ten znak przecież zmieniliśmy.
Po wykonaniu zlecenia odpowiedzią na "%" będzie już zwyczajne "ą". Co prawda zabieg się udał, nie ma jednak wielkich powodów do dumy. Przeprowadziliśmy go dość prymitywnie. Jeśli zamierzamy modelowaniem znaków zajmować się częściej, nie od rzeczy będzie sporządzenie prostego programu narzędziowego do edycji. Program taki powinien wyświetlać w powiększeniu dowolny spośród dostępnych znaków, pozwolić na jego wygodną modyfikację metodą naprowadzania kursora na poszczególne pola, po czym zapisać w pamięci znak w skorygowanej formie.
Wyświetlenie powiększonego znaku nie sprawi nam trudności (patrz program 30):

Program 37a

```
200 CLS
210 INPUT "Korygowany symbol: "; a$
220 LET A=FN W(a$)
230 FOR I=0 TO 7
240   LET bajt=PEEK (A+I)
250   FOR B=0 TO 7
260     LET kolor=6-3*FN T(bajt,B)
270     PRINT AT 8+I, 19-B; PAPER kolor; " ";
280   NEXT B
290 NEXT I
```

Zamiast najpierw umieszczać znak na ekranie, a dopiero potem analizować go punkt po punkcie funkcją **POINT**, wybraliśmy drogę bezpośrednią. Obliczamy adres pierwszego bajtu wzorca znaku i przypisujemy go zmiennej **A**. Następnie pobieramy po jednym wszystkie 8 bajtów wzorca, przypisując je zmiennej **bajt**. Analizujemy bajt bit po bicie funkcją **FN T**. Zależnie od tego, czy wartość bitu wynosi Ø czy 1, dobieramy barwę odpowiadającego mu pola w powiększonym rysunku znaku. Dla bitu zerowego wartość **FN T(bajt, B =Ø)**, a więc wartością zmiennej **kolor** będzie 6 (żółty). Jeśli bit = 1, **FN T (bajt, B)** = 1, a zmienna **kolor** stanie się równa 3 (magenta, purpura). W końcu w odpowiednim polu kwadratu 8×8 pośrodku ekranu wyświetlamy puste pole barwy żółtej lub purpurowej. Po powtórzeniu tego procesu dla wszystkich bitów każdego z 8 bajtów wzorca, na ekranie ukaże się powiększony obraz znaku.
Znak rysuje się wolno. Winę ponosi funkcja **FN T**, a zwłaszcza występujące tu potęgowanie. Jak widać, nie zawsze BASIC jest dostatecznie efektywnym językiem programowania.
Zadanie najtrudniejsze: edycja znaku. Posłużmy się ruchomym kursorem--znacznikiem. Powinien on pozwalać przemieszczać się w środkowym

kwadracie i naprowadzać na każde z 64 pól. Po naprowadzeniu musi istnieć możliwość zapalania lub gaszenia wskazanego znacznikiem pola. Przemieszczać kursor najwygodniej klawiszami "5"—"8", na których naniesiono strzałki w czterech kierunkach. Wciskając "1" spowodujemy zapalenie, "Ø" — wygaszenie pola. Pozostaje pytanie, jak zobrazować znacznik. Musi on być widoczny zarówno w zapalonych, jak i w wygaszonych polach. Nie może przy tym utrudniać obserwacji pola, w którym się znajduje. Najprościej będzie podświetlić pole ze znacznikiem, ustawiając w nim atrybut **BRIGHT =1**. Wciskając spację zakończymy edycję.
Aktualne położenie znacznika wskazywać będą zmienne **X** (kolumna) i **Y** (wiersz). Początkowo zmienne te wskazują na lewy górny róg kwadratu.

Program 37b

```
300 LET X=12: LET Y=8
310 GO TO 340
320 IF X=X0 AND Y=Y0 THEN GO TO 360
330 POKE FN A(Y0,X0),PEEK FN A(Y0,X0)-64
340 POKE FN A(Y,X),PEEK FN A(Y,X)+64
350 LET X0=X: LET Y0=Y
360 LET K$=INKEY$
370 IF K$="5" AND X>12 THEN LET X=X-1
380 IF K$="8" AND X<19 THEN LET X=X+1
390 IF K$="6" AND Y<15 THEN LET Y=Y+1
400 IF K$="7" AND Y>7 THEN LET Y=Y-1
410 IF K$="1" THEN POKE FN A(Y,X),BIN 01011000
420 IF K$="0" THEN POKE FN A(Y,X),BIN 01110000
430 IF K$<>" " THEN GO TO 320
```

Pole z kursorem ma zostać podświetlone. Funkcja **FN A** ustala adres bajtu atrybutów tego pola. Do aktualnej wartości tego pola dodajemy 64 (**BIN 01ØØØØØØ**), ustawiając odpowiedzialny za jaskrawość bit nr 6 na 1 bez zmiany innych atrybutów (linia 34Ø). Współrzędne rozjaśnianego pola przechowują zmienne **X Ø, Y Ø**. Dalsze działania zależeć będą od manipulacji przy klawiaturze. Funkcja **INKEY$** odpytuje stan klawiatury. Znak, odpowiadający wciśniętemu klawiszowi, przechowywany jest przez **K$**. Zmienna ta jest porównywana ze wszystkimi dopuszczalnymi znakami sterującymi: "5"—"8", "1", "Ø" i spacją. W razie wykrycia zgodności program podejmie odpowiednie działania. Jeśli np. wciśnięto klawisz ze strzałką w lewo, tzn. "5", a kursor nie znajduje się przy lewej krawędzi kwadratu (bada to warunek **X>12**), wtedy aktualna współrzędna kursora **X** zmniejszy się o 1, co odpowiada przesunięciu kursora o jedno pole w lewo. Podobnie postępuje się po wciśnięciu innych klawiszy kierunkowych.
"1" oznacza zapalenie pola. Trzeba wpisać odpowiednie atrybuty: atrament dowolny, tło=3, jaskrawość=1 (w tym polu jest znacznik), migotanie=Ø.

Analogicznie, skasowanie pola (klawisz "Ø") powoduje wpisanie atrybutów: tło=6 (barwa pól pustych), jaskrawość=1, migotanie=Ø. Jeśli wprowadziliśmy spację, w wyniku wykonania instrukcji 43Ø opuścimy program edycji znaku, przechodząc do instrukcji o wyższych numerach. W przeciwnym razie — niezależnie od tego, czy wciśnięto inny klawisz lub nie naciśnięto żadnego — nastąpi powrót do linii 32Ø. Pamiętamy, że klawiszami kierunkowymi zmienialiśmy tylko wartości **X** i **Y**. Nie oznaczało to jednak automatycznego przemieszczania znacznika na ekranie. Musimy się o to zatroszczyć sami. Najpierw sprawdzamy, czy znacznik w ogóle się przemieścił. Rozpoznamy to po niezgodności aktualnych współrzędnych z poprzednimi. W takim razie wykonamy dwie czynności:

— Przywrócimy poprzedni stan pola, które znacznik opuszcza (**X Ø**, **Y Ø**). Odejmując 64 od wartości atrybutu zerujemy bit jaskrawości nie wpływając na pozostałe.

— Rozjaśnimy pole nowej lokalizacji znacznika.

Gdy znak jest zredagowany, należy przenieść go z ekranu do pamięci wzorców. Osiągniemy to analizując kolejne wiersze kwadratowego pola na ekranie i budując na tej podstawie bajty wzorca:

Program 37c

```
500 FOR I=Ø TO 7
510   LET bajt=Ø
520   FOR B=Ø TO 7
530     IF FN P(3+I,19-B)=3 THEN LET bajt=bajt+2^B
540   NEXT B
550   POKE A+I, bajt
560 NEXT I
570 GO TO 2ØØ
```

Zmienna **I** przedstawia numer kolejnego wiersza znaku i zarazem numer bajtu, zmienna **B** — numer bitu. Kolor tła w polu badamy funkcją **FN P**. Jeśli jest ono zielone (=3), dostawiamy odpowiedni bit przez dodanie odpowiedniej potęgi liczby 2 (patrz: Rozwinięcie liczby dwójkowej). Skompletowany bajt zapisujemy do bloku wzorców. Po przesłaniu do pamięci wszystkich ośmiu bajtów program pyta o kolejny znak do modyfikacji.

Nasz program pozwala na projektowanie zarówno znaków należących do głównego generatora (o ile przenieśliśmy go do RAM!), jak i UDG. Istotnym ulepszeniem byłoby np. zespolenie programów 36 i 37.

Mozolnie opracowany zestaw znaków należy zapisać na taśmie magnetofonowej. W przypadku UDG osiągniemy to zleceniem:

SAVE "UDG" CODE USR "a", 168

Zawartość głównego generatora znaków zapiszemy zleceniem:

SAVE "ZNAKI" CODE FN D(236Ø6)+256, 768

Raz przygotowane znaki można użyć w różnych programach. Ponowne załadowanie UDG jest proste:

LOAD "UDG" CODE

Jeśli obszar UDG zlokalizowany jest gdzie indziej niż w fazie redagowania (np. przygotowany był na Spectrum 48 KB, wczytywany jest do wersji 16 KB), użyjemy zlecenia:

LOAD "UDG" CODE USR "a"

Ładowanie generatora głównego jest bardziej złożone. Robimy to zwykle po ponownym włączeniu komputera. Zacząć trzeba więc od wykrojenia niezbędnego obszaru pamięci instrukcją **CLEAR**:

CLEAR USR "a" - 769
LOAD "ZNAKI" CODE USR "a" - 769

Jeśli ładujemy generator znaków w to samo miejsce, z którego zapisaliśmy go na taśmę, wystarczy zlecenie:

LOAD "ZNAKI" CODE

Po załadowaniu nowych wzorców wystarczy zmienić wartość **CHARS**:

LET adr=236Ø6:LET db=USR "a"-256 : GO SUB 989Ø

Jest prostsze wyjście. Przed zapisaniem generatora na taśmę badamy obydwa bajty **CHARS**:

PRINT PEEK 236Ø6, PEEK 236Ø7

Wartości zapisujemy, a po załadowaniu generatora wpisujemy ponownie dwoma instrukcjami **POKE**. Generator znaków zlokalizować można oczywiście w innym obszarze RAM. Trzeba tylko unikać kolizji z językiem BASIC poprzez użycie na początku instrukcji **CLEAR** z odpowiednim argumentem.

Pismo plakatowe

Przy redagowaniu tekstów zamiast 32 znaków w linii lepiej byłoby mieć ich 42 lub nawet 64. Wystarczyłoby zdefiniować własne wzorce znaków w prostokącie 6×8 lub 4×8 punktów i ułożyć program, który wrysuje je w ekran. W języku BASIC program taki działałby bardzo wolno.
Od czasu do czasu przydałoby nam się też pismo powiększone. Można je zastosować w nagłówku programu (lub przy korepetycjach z arytmetyki), w którym zadanie "2 + 2 = ?" nie było skryte w kącie ekranu, lecz zajmowało połowę jego powierzchni.
Stworzymy uniwersalną procedurę rysującą znaki. Niech lokalizacja znaku

będzie dowolna z dokładnością do punktu na ekranie, a powiększenie pozwala nastawiać się niezależnie w obu osiach.
Aby powiększyć znak w osi X 2 razy, w osi zaś Y — 3 razy, należy pojedynczy punkt siatki znaku zastąpić prostokątem 2×3 punkty. Niech **X Ø** i **Y Ø** wyznaczają współrzędną lewego, górnego pola znaku. Niech **powx** i **powy** przedstawiają powiększenie w osi X i Y. Niech **Z$** zawiera znak do umieszczenia na ekranie. Zmienna **U** odliczać będzie poszczególne pola w poziomie od Ø do 7, zmienna **V** to samo w pionie.
Najpierw napiszemy podprogram rysujący na ekranie wypełniony prostokąt o bokach długości **powx** i **powy**. Podprogram ten będziemy wykorzystywać zamiast instrukcji **PLOT** do nanoszenia poszczególnych punktów znaku. Lokalizacja prostokąta zależna będzie od **X Ø** i **Y Ø** oraz **U** i **V**.

Program 38

```
9370 REM PR(XØ, YØ, U, V, powx, powy)
9371 FOR q=Ø TO powx-1
9372   PLOT XØ+U*powx+q, YØ-V*powy
9373   DRAW Ø, 1-powy
9374 NEXT q
9376 RETURN
```

Instrukcja **PLOT** zapala kolejno punkty górnej krawędzi prostokąta. Jeśli **powy** wynosi więcej niż 1, czyli wysokość prostokąta jest większa niż jeden punkt, instrukcja **DRAW** prowadzi w dół odcinek długości **powy−1**.
Zorganizowanie właściwego wyprowadzania znaku nie sprawi już trudności. Funkcją **FN W** wyznaczymy adres wzorca znaku. Pobierając kolejne bajty i badając ich bity, będziemy wypełniali — lub nie — odpowiednie pola siatki znaku:

Program 39

```
9360 REM ZN(XØ, YØ, powx, powy, Z$)
9361 LET q1=FN W(Z$)
9362 FOR V=Ø TO 7
9363   LET q2=PEEK (q1+V)
9364   FOR U=Ø TO 7
9365     IF FN T(q2,7-U)=1 THEN GO SUB 9870
9366   NEXT U
9367 NEXT V
9368 RETURN
```

Nadszedł czas próby. Napiszemy instrukcje nadające wartość parametrom:

Program 40

```
10 INPUT "Znak: "; Z$
20 LET X0=50: LET Y0=160
30 LET powx=3: LET powy=5
40 GO SUB 9860
50 STOP
```

Jeśli znak ma być przedstawiony inwersyjnie, wystarczy w instrukcji warunkowej (wiersz 9865) porównać wartość bitu (**FN T**) nie z 1, lecz z Ø.
Aczkolwiek program działa, szybkość pracy nas nie zadowala. Kreślenie jednego znaku trwa co najmniej kilkanaście sekund. Można się na to zgodzić przy wypisywaniu nagłówka. W większości zastosowań procedura będzie jednak zbyt wolna.
Małą prędkość pracy programów w języku BASIC obserwowaliśmy już poprzednio. Straty czasu powodowane są zazwyczaj przez zagnieżdżone pętle, wykonujące bardzo proste operacje. Radykalnym rozwiązaniem problemu szybkości jest sięgnięcie po procedury napisane w języku maszynowym.

Elementarz języka maszynowego

Programowanie w języku maszynowym nie jest tajemną wiedzą, zastrzeżoną dla profesjonalistów. Podstawy tej umiejętności opanuje każdy, kto potrafi logicznie myśleć, jest cierpliwy i systematyczny.
Pełny wykład z języka maszynowego byłby tu zbyt obszerny. Znajomość elementów języka wewnętrznego ułatwi nam jednak rozumienie prostych programów maszynowych i wykorzystanie ich we własnych programach w języku BASIC.
Program maszynowy to jedyna postać programu, którą mikroprocesor może wykonać bezpośrednio. Program w języku BASIC zapisany jest w pamięci w postaci symbolicznej. Jego wykonaniem zajmuje się obszerny program maszynowy, zawarty w pamięci ROM i zwany interpreterem języka BASIC. To właśnie ten program wykonuje się bezustannie w normalnych warunkach, analizując znak po znaku program w BASIC-u, rozpoznając i wykonując opisane w nim czynności. Wykonanie jednej instrukcji języka BASIC wymaga zazwyczaj kilkuset do kilku tysięcy rozkazów maszynowych. Rozkazy te w większości nie służą właściwemu przetwarzaniu informacji, lecz celom administracyjnym: rozpoznają instrukcje i badają ich poprawność, wyszukują w pamięci zmienne itp. Z tego m.in. powodu programy w języku BASIC działają wolno.
Mikroprocesor operuje nie na zmiennych wskazywanych przez nazwę, lecz na zawartości pojedynczych komórek pamięci o ściśle określonych adresach.

Operacje wykonywane są na bajtach, w najlepszym razie na dwubajtowych liczbach całkowitych.
Program maszynowy nie jest czytelnym tekstem rozbitym na numerowane linie, lecz ciągiem bajtów, umieszczonych w wyznaczonych, przeważnie kolejnych komórkach pamięci. Zapisując program maszynowy np. na kartce, stosujemy zwykle tzw. język symboliczny, potocznie zwany assemblerem. Ułatwia to rozumienie programu. Przed wprowadzeniem programu do maszyny trzeba zapis symboliczny przetłumaczyć na odpowiadający mu ciąg bajtów. W języku symbolicznym jednej instrukcji odpowiada dokładnie jeden rozkaz maszynowy. Tłumaczenie można wykonać ręcznie. Przy programie dłuższym niż kilkanaście instrukcji postępowanie takie jest żmudne i podatne na błędy. Używamy wtedy programu tłumaczącego zapis symboliczny na kod maszynowy, zwanego właśnie assemblerem.
Mikroprocesor Z80 zawiera kilkanaście komórek jedno- lub dwubajtowych, zwanych rejestrami wewnętrznymi. Uczestniczą one w większości operacji. Rozkazy maszynowe działają na określonych rejestrach, nie są tu więc potrzebne adresy, jak w pamięci. Wystarczy nam znajomość siedmiu jednobajtowych rejestrów, oznaczanych umownie literami A, B, C, D, E, H, L. Największe możliwości posiada rejestr A, zwany akumulatorem. Rejestry B i C, D i E oraz H i L można łączyć w pary przechowujące dwubajtowe liczby całkowite. Przeanalizujmy prościutki program maszynowy, składający się zaledwie z trzech rozkazów:

kod maszynowy	zapis symboliczny	komentarz
62	**LD A, 255**	; Do rejestru A wpisz stałą wartości 255_{10}
255		
5Ø	**LD 16385, A**	; Zawartość rejestru A prześlij do komórki RAM o adresie 16384
1-64		
2Ø1	**RET**	; Wróć do programu, który wywołał tę procedurę

Rozkazy w Z80 składają się z 1 do 4 bajtów. Pierwszy bajt jest najczęściej tzw. kodem operacji, określającym rodzaj czynności. Gdy potrzebne są dodatkowe informacje, zawarte są w jednym lub dwóch bajtach następnych. Kodem operacji pierwszego rozkazu jest 62_{10}. Mikroprocesor rozpozna ją jako polecenie pobrania bajtu następującego za kodem operacji i wpisania jego wartości do rejestru A. Wykonawszy pierwszy rozkaz mikroprocesor odczytuje kolejny bajt programu i znów interpretuje go jako kod operacji. Liczba $5Ø_{10}$ w wewnętrznym słowniku mikroprocesora jest rozkazem skopiowania rejestru A do komórki pamięci o podanym adresie. Adres komórki pamięci jest dwubajtowy. Za kodem operacji następuje więc kolejno młodszy i starszy bajt adresu. W naszym przypadku 1+256•64=16385. Jest to adres komórki

w pamięci ekranu. W chwili wykonania rozkazu zawartość A = 255, taka sama wartość zostanie wpisana więc do komórki nr 16385.
Dwa wykonane do tej pory rozkazy maszynowe odpowiadają funkcjonalnie instrukcji **POKE 16385, 255**. Ich wykonanie trwa jednak kilkaset razy krócej. Narzuca się pytanie, czy zamiast dwóch rozkazów nie można by użyć pojedynczego rozkazu maszynowego. Otóż nie. W repertuarze Z80 nie ma rozkazu zapisu stałej do komórki o podanym adresie! Sytuacja taka jest typowa. Proste i zwięźle zapisane w języku BASIC operacje wymagać będą w języku maszynowym od kilku do kilkuset rozkazów.
Ostatni rozkaz o kodzie 201_{10} (**RET**) jest jednobajtowy i składa się z samego kodu operacji. Mówiąc w uproszczeniu, rozkaz ten powoduje powrót mikroprocesora do wykonywania programu w języku BASIC (program maszynowy wywoływany jest z języka BASIC jako podprogram. **RET** spełnia tu podobną rolę jak **RETURN** w BASIC-u).
Zauważmy, że w zapisie symbolicznym dla Z80 najpierw podajemy, dokąd informację przesłać, a dopiero potem, skąd ją pobrać.
Przedstawiona procedura maszynowa jest przesuwna (przemieszczalna, relokowalna). Znaczy to, że będzie ona działać w dowolnym obszarze pamięci. Niestety, większość programów maszynowych, w tym wszystkie bardziej złożone, wymagają umieszczenia w ściśle określonym miejscu pamięci.
Wpiszmy nasz programik do pamięci i spowodujmy jego wykonanie. Dzięki przesuwności możemy wpisać go w dowolne miejsce, np. w bufor drukarki (23296—23551). Użyjemy instrukcji **POKE**:

Program 41

```
10 FOR A=23296 TO 23301
20   READ X: POKE A, X
30 NEXT A
40 DATA 62,255,50,1,64,201
```

Umieszczanie wartości kolejnych bajtów programu w zbiorze **DATA** i kolejne ich odczytywanie, i zapis do pamięci w pętli **FOR... NEXT** to często stosowany sposób ładowania do pamięci krótkich programów maszynowych. Do uruchamiania programu maszynowego służy w Spectrum funkcja **USR**. Jej argumentem jest adres rozkazu, od którego należy rozpocząć realizację programu maszynowego. Żądanie obliczenia wartości funkcji **USR** powoduje za każdym razem wykonanie procedury maszynowej pod wskazanym adresem.
Pozornie dziwne rozwiązanie, polegające na uruchomieniu programu maszynowego wywołaniem funkcji, ma uzasadnienie. Procedury maszynowe wykonują często operacje, których wynik należy przekazać do programu w języku BASIC. Funkcja **USR** dostarcza wartości: w tym wypadku będzie to liczba całkowita z zakresu Ø—65535, przedstawiająca zawartość pary

rejestrów BC procesora w chwili wykonania rozkazu **RET** (rejestr B — bajt starszy). Skasujmy ekran przez **CLS** i wykonajmy zlecenie:

LET X=USR 23296

U góry ekranu w drugiej kolumnie pojawi się czarna kreska. Wartość zmiennej X nie interesuje nas. Tym razem instrukcja przypisania była tylko pretekstem użycia **USR** (w chwili wywołania procedury maszynowej zawartość pary BC odpowiada adresowi wywołania). Zamiast **LET** można użyć **RANDOMIZE**, o ile nie koliduje to z jej właściwym przeznaczeniem:

RANDOMIZE USR 23296

Program maszynowy w pamięci można modyfikować. Chcemy oto wpisać wartość nie 255, lecz 199 i to do komórki nie 16385, lecz 16395. Instrukcja: **LD A, 255** ustąpi miejsca **LD A, 199**. Kod operacji pozostanie oczywiście nie zmieniony. Zmodyfikujemy tylko drugi bajt-argument:

POKE 23297, 199

Podobnie w drugim rozkazie. Rodzaj operacji nie zmienia się — kod operacji pozostaje. Modyfikacji ulegnie tylko młodszy bajt adresu:

POKE 23299, 11

Po uruchomieniu przez **RANDOMIZE USR 23296** program wpisze w ekran nie kreskę, lecz dwa krótkie odcinki (199_{10} = BIN 11ØØØ11) w dwunastej kolumnie.

Pierwszy programik był tylko wprawką. Przyjrzyjmy się zatem procedurze bardziej złożonej, lecz posiadającej praktyczną przydatność. Jej zadaniem będzie wypełnianie pamięci atrybutów bajtem o zadanej wartości, np. **71 = BIN Ø1ØØØ111** (tło czarne, atrament biały, jaskrawość podwyższona).

kod maszynowy	zapis symboliczny	komentarz
33, Ø, 88	**LD HL, 22528**	; adres początku pamięci atrybutów do pary HL
1, Ø, 3	**LD BC, 768**	; ilość bajtów w pamięci atrybutów do pary BC
54, 71	**LD (HL), 71**	; wyślij stałą do komórki o adresie zawartym w parze HL
35	**INC HL**	; inkrementuj (zwiększ o 1) zawartość pary HL
11	**DEC BC**	; dekrementuj (zmniejsz o 1) zawartość pary HL
12Ø	**LD A, B**	; zawartość rejestru B wpisz do A
177	**OR C**	; oblicz sumę logiczną rejestrów A i C

```
32, 248     JR NZ, $-8      ; jeśli wynik niezerowy, skocz
                            wstecz o 8 komórek
201         RET             ; powrót do programu wywołu-
                            jącego
```

Rozkaz **LD HL, 22528** traktuje rejestry H i L jako pojedynczy rejestr 16-bitowy. Umieszczona w nim liczba jest adresem pierwszej komórki bloku atrybutów (0+256*88 =22528). Podobnie rozkaz **LD BC, 768** ładuje do rejestrów B i C, traktowanych jako całość, liczbę 768. Rozkaz **LD (HL), 71** (kod operacji 54) powoduje zapisanie do wybranej komórki pamięci stałej zawartej w drugim bajcie (w naszym przypadku: 71). Adres komórki pamięci podany jest pośrednio: jest nim liczba aktualnie zawarta w parze HL. Ponieważ w parze tej jest aktualnie 22528, do komórki o tym adresie wpisana zostanie stała.

Jednobajtowy rozkaz **INC HL** zwiększa o 1 liczbę zawartą w HL (było 22528, jest 22529). Rozkaz **DEC BC** zmniejsza o 1 zawartość BC (było 768, jest 767).

Rozkaz **LD A, B** wpisuje do A zawartość rejestru B, zaś **OR C** wyznacza sumę logiczną rejestru C i rejestru A. Suma logiczna wykonywana jest oddzielnie na każdej parze odpowiadających sobie bitów. Wynik operacji będzie zerem tylko wtedy, gdy oba operandy (liczby uczestniczące w operacji) także będą zerowe. Ponieważ do A wpisaliśmy uprzednio zawartość B, rozkaz **OR C** sprawdza faktycznie, czy liczba zawarta w parze BC jest zerem. Rozkaz **JR NZ, $-8** to skok warunkowy. Warunkiem wykonania skoku jest niezerowy wynik poprzedniej operacji (**JR NZ** — Skocz, jeśli nie zero). Skok nastąpi wstecz o 8 komórek, czyli do rozkazu **LD (HL), 71**.

Rozkaz skoku warunkowego będzie wykonywał się tak długo, aż liczba zawarta w BC stanie się zerem. Utworzyliśmy więc pętlę w programie maszynowym. Za każdym powtórzeniem pętli zawartość BC zmniejsza się o 1. Pętla wykona się więc dokładnie 768 razy. Za każdym powtórzeniem zwiększana jest także zawartość pary HL. Stała 71 zostanie więc wpisana do 768 kolejnych komórek pamięci RAM, poczynając od adresu 22528. W języku BASIC działanie programu zilustrować można by następująco:

Program 42

```
10 LET HL=22528
20 LET BC=768
30 POKE HL, 71
40 LET HL=HL+1
50 LET BC=BC-1
60 IF BC<>0 THEN GO TO 30
```

Zmienne **HL** i **BC** nie mają oczywiście nic wspólnego z rejestrami procesora. Uruchommy ten program. Jego wykonanie zajęło ok. 12 sekund. Wypróbuj-

my teraz jego maszynowy odpowiednik. Na szczęście i on jest przesuwny. Umieścimy go znowu w buforze drukarki:

Program 43

```
100 FOR A=23296 TO 23310
110   READ X: POKE A, X
120 NEXT A
130 DATA 33,0,88,1,0,3,54,71,35
140 DATA 11,120,177,32,248,201
```

Uruchommy program ładujący. Zleceniem **LIST** wylistujmy zawartość pamięci i poprzez **RANDOMIZE USR 23296** uruchommy procedurę maszynową. Tym razem wygląd ekranu zmienił się w jednej chwili. Wykonanie programu trwało ok. 10 milisekund.
Nasz program jest uniwersalny. Może być zastosowany do wypełniania dowolną zawartością wybranego obszaru pamięci RAM. Wystarczy zmienić adres początkowy, liczbę bajtów do zapisania i wartość stałej — wypełniacza. Spróbujmy np. zapisać całą pamięć ekranu bajtem $ØØ11ØØ11_2 = 51_{10}$. Musimy zapisać nowy argument instrukcji **LD HL, stała**. Możemy posłużyć się podprogramem **DPOKE** (program 22):

LET adr=23297:LET DB=16384 : GO SUB 989Ø

lub "ręcznie" rozbić argument na bajt mniej i bardziej znaczący:

POKE 23297, Ø : POKE 23298, 64

Podobnie postąpimy z instrukcją **LD BC, stała**. Wartością stałej będzie teraz 6144:

POKE 233ØØ, Ø : POKE 233Ø1, 24

Zmienimy jeszcze stałą, stanowiącą argument instrukcji **LD (HL), stała**:

POKE 233Ø3, 51

Uruchamiając ponownie zmodyfikowany program przez **RANDOMIZE USR 23296**, otrzymamy skutek inny niż poprzednio. Także i tym razem wydarzenia rozegrały się błyskawicznie.

Zapis szesnastkowy: niepotrzebna komplikacja czy wymóg ekonomiki?

Załadowanie do pamięci kolejnych bajtów programu zapisanych w zbiorze **DATA** jest proste. Wystarczy zwykła pętla. Nierzadko zdarza się jednak, że ładowany program maszynowy liczy kilkaset lub więcej bajtów. Zapis danych

w postaci stałych dziesiętnych okazuje się wtedy niestety bardzo nieekonomiczny. Jedna wartość pochłania ok. 9 bajtów programu w języku BASIC. Przepisując program np. z książki łatwo popełnić też przypadkowy błąd. Jego wykrycie w masie danych może kosztować sporo czasu i nerwów. Czas zająć się ekonomiczną i niezawodną metodą wprowadzania programów maszynowych do pamięci operacyjnej.
Zamiast zapisu dziesiętnego lub dwójkowego zastosujemy notację szesnastkową, zwaną też heksadecymalną (właściwie: sedecymalną). Każdy bajt zakodujemy dwiema cyframi szesnastkowymi.
W systemie o podstawie 16 potrzeba nam aż 16 różnych cyfr. Zbiór dziesięciu prawdziwych cyfr Ø—9 uzupełnimy więc sześcioma pierwszymi literami alfabetu: A—F. Wartości, przypisane tym znakom, są następujące:

Cyfra szesnastkowa	Wartość dziesiątkowa	Wartość dwójkowa
Ø	Ø	ØØØØ
1	1	ØØØ1
2	2	ØØ1Ø
3	3	ØØ11
4	4	Ø1ØØ
5	5	Ø1Ø1
6	6	Ø11Ø
7	7	Ø111
8	8	1ØØØ
9	9	1ØØ1
A	1Ø	1Ø1Ø
B	11	1Ø11
C	12	11ØØ
D	13	11Ø1
E	14	111Ø
F	15	1111

Tabl. 1

Jedna cyfra szesnastkowa odpowiada czterem kolejnym bitom:

$$111Ø1ØØ1_2 = E9_{16}$$

Przejście od zapisu dwójkowego do szesnastkowego i odwrotnie jest proste. Wystarczy czterobitowym grupom przyporządkować odpowiednie cyfry szesnastkowe. Do zapamiętania jednego bajtu w postaci szesnastkowej potrzeba nam zaledwie dwóch znaków i tyleż bajtów pamięci programu w języku BASIC. Przed wykorzystaniem musimy wszakże przekształcić liczbę szesnastkową na dziesiętną. Możliwym rozwiązaniem jest utworzenie zbioru znaków wzorcowych, zawierającego wszystkie cyfry szesnastkowe ułożone

kolejno. Tłumaczoną cyfrę szesnastkową porównywalibyśmy z kolejnymi znakami tego łańcucha. Po wykryciu zgodności, na podstawie numeru pozycji wzorca w łańcuchu, określalibyśmy wartość cyfry:

Program 44

```
10 INPUT "Cyfra szesnastkowa: "; H$
20 FOR I=0 TO 15
30   IF H$<>"0123456789ABCDEF"(I+1) THEN NEXT I
40 PRINT "Cyfra odpowiada liczbie "; I
```

Przy kolejnym tłumaczeniu setek i tysięcy cyfr szesnastkowych ważna będzie szybkość pracy. Nie sprzyja jej wielokrotnie powtarzana pętla. Wykorzystajmy więc znajomość kodu znaków:

Program 45

```
10 INPUT "Cyfra szesnastkowa: "; H$
20 LET I=CODE H$-48-7*(H$>="A")
40 PRINT "Cyfra odpowiada liczbie "; I
```

Liczba 48 to kod cyfry "0". Cyfra "1" ma kod 49 itd. aż do "9" — kod 57. Po odjęciu od kodu cyfry liczby 48 otrzymamy odpowiadającą cyfrze liczbę. Kod litery "A" to 65, "B" — 66, "F" — 70. Gdyby cyfra była znakiem "A" ÷ "F", musimy odjąć jeszcze 7. W sumie odjęliśmy 48 + 7 = 55. 65 − 55 = 10 — liczba odpowiadająca "A" itd.

Jak zabezpieczyć się przed błędami przy przepisywaniu programów?

Wszystkie programy maszynowe, przedstawione w tej książce, zapisano następująco. Grupa bajtów przedstawiona jest jako zwarty łańcuch cyfr szesnastkowych — po dwie cyfry na bajt. Wszystkie bajty tworzące tę grupę są sumowane. Reszta z dzielenia tej sumy przez 256 tworzy tzw. sumę kontrolną, dopisaną do łańcucha jako 2 ostatnie bajty. Wyobraźmy sobie, że przepisując, przypadkowo zmienimy wartość jakiejś cyfry lub zamienimy miejscami dwa sąsiednie znaki. Program ładujący odczytuje wartości kolejnych bajtów i w analogiczny sposób oblicza ich sumę, jak było to robione przy opracowaniu programu do druku. Reszta z dzielenia obliczonej sumy przez 256 porównywana jest z ostatnim, kontrolnym bajtem. Jeśli suma obliczona nie zgadza się z odczytaną, z pewnością popełniliśmy gdzieś błąd. Program ładujący sygnalizuje nam numer błędnego wiersza. Ułatwia to lokalizację przekłamania.

Przed właściwym blokiem bajtów programu znajdują się dwie stałe dziesiętne. Pierwsza podaje, od którego wiersza programu zaczyna się blok danych (na wypadek błędu), druga zaś jest adresem pierwszego bajtu w pamięci. Koniec bloku danych rozpoznaje program po pustym łańcuchu " ". Aby

załadować program maszynowy do pamięci, wystarczy po prostu wykonać instrukcję **RESTORE** z argumentem wskazującym na pierwszy wiersz bloku danych i uruchomić program ładujący zleceniem **GO SUB 9990**. W ten sposób można załadować do różnych obszarów pamięci kilka oddzielnych programów maszynowych. A oto właściwy program ładujący:

Program 46

```
9990 REM LOADER PROGRAMOW MASZYNOWYCH
9991 READ A,N
9992 READ L$: LET L=LEN L$: LET S=0: LET K=2: LET N=N+1
9993 IF L=0 THEN RETURN
9994 LET Q2=CODE L$(K-1): LET Q1=CODE L$(K)
9995 LET C=Q1-48-7*(Q1>64)+16*(Q2-48-7*(Q2>64))
9996 IF K<L THEN POKE A,C: LET S=S+C: LET K=K+2: LET A=A+1: GO TO 9994
9997 IF S-256*INT (S/256)<>C THEN PRINT "BLAD w linii ";N: STOP
9998 GO TO 9992
```

Pożyteczne procedury maszynowe

Nie pozostaje nam nic innego, jak zrobić użytek z naszego programu ładującego (ang. loader). Poniżej zamieszczono kilka krótkich, lecz często przydatnych podprogramów maszynowych. Zawarte są one w postaci szesnastkowej w zbiorze **DATA**.

Wszystkie przedstawione poniżej procedury są relokowalne (przesuwne), tzn. mogą być załadowane i uruchomione bez dodatkowych modyfikacji w dowolnym obszarze pamięci RAM. Pierwotnie przewidywano umieszczenie procedur począwszy od komórki 64ØØØ. Ich adresy wybrano w tym obszarze tak, by nie kolidowały ze sobą. Przy korzystaniu z procedur zlokalizowanych we wspomnianym obszarze trzeba oczywiście przed ich ładowaniem wykonać zlecenie **CLEAR 63999**. Gdy zajdzie potrzeba przemieszczenia procedur w inne miejsce pamięci (w wersji 16 KB, przy współpracy z innymi programami maszynowymi itp.), wystarczy zmodyfikować adres ładowania w pierwszej linii bloku danych każdej procedury. Adres pierwszego bajtu każdej procedury (jest to właśnie adres ładowania) oznaczać będziemy umownie przez **adrp**.

Wygodne może okazać się oddzielne nagranie na taśmę bloków danych wszystkich procedur i programu ładującego. W razie potrzeby można je włączać do własnych programów zleceniem **MERGE**.

1. **Rotacja zawartości ekranu w prawo** (23 bajty, **adrp**=64ØØØ)

Procedura przemieszcza całą zawartość ekranu (24 linie) o 1 punkt w prawo. Atrybuty nie są przesuwane (przy przemieszczeniu o 1 punkt nie ma to sensu). Punkty opuszczające ekran z prawej strony pojawiają się ponownie z lewej. Wielokrotne wywołanie procedury w pętli powoduje wrażenie

przewijania się ekranu w prawo. Instrukcją **POKE adrp+13, 175** (np. **POKE 64Ø13, 175**) modyfikujemy procedurę tak, że opuszczająca ekran informacja nie powraca z lewej, na jej miejsce wstawiane są puste punkty. Przywrócenie stanu wyjściowego: **POKE adrp+13, 31**. Po instrukcji **POKE adrp+4, 128** przewijane są tyko górne 2/3 ekranu, a po **POKE adrp+4, 64** — górna 1/3. **POKE adrp+2, 72 : POKE adrp+4, 64** powoduje przewijanie wyłącznie środkowego pasa, a **POKE adrp+2, 8Ø; POKE adrp+4, 64** — pasa dolnego. Adres wywołania: **adrp**. Przykład użycia (dwukrotne przewinięcie całego ekranu):

FOR I=1 TO 512: RANDOMIZE USR 64ØØØ : NEXT I

Program 47

```
9600 REM Rotacja ekranu w prawo
9601 DATA 64000,9601
9602 DATA "2100400EC00620E5DD17"
9603 DATA "E1DD7E1F1FCB1E231096"
9604 DATA "FB0D20EFC9E0",""
```

2. **Rotacja zawartości ekranu w lewo** (23 bajty, **adrp**=64Ø23)
Działanie podobne do procedury poprzedniej. Ekran przewijany jest za każdym wywołaniem o 1 punkt w lewo. Aby uniknąć powracania punktów opuszczających ekran z lewej, wystarczy **POKE adrp+13, 175**. Odtworzenie stanu pierwotnego: **POKE adrp+13, 23**. Przesuwanie dolnych 2/3 ekranu: **POKE adrp+4, 128**. Przesuwanie dolnej 1/3: **POKE adrp+4, 64**. Przesuwanie pasa środkowego: **POKE adrp+2, 79 : POKE adrp+4, 64**. Przesuwanie pasa górnego: **POKE adrp+2, 71 : POKE adrp+4, 64**. Przykład użycia (przewinięcie środkowego pasa ekranu):

POKE 64Ø25, 79 : POKE 64Ø27, 64 : FOR I= TO 256 :
RANDOMIZE USR 64Ø25 : NEXT I

Program 48

```
9610 REM Rotacja ekranu w lewo
9611 DATA 64023,9611
9612 DATA "21FF570EC00620E5DD2D"
9613 DATA "E1DD7EE117CB162B1050"
9614 DATA "FB0D20EFC9E0",""
```

3. **Płynny scrolling ekranu w górę** (28 bajtów, **adrp**=64Ø46)
Procedura powoduje przesuw zawartości ekranu w górę o 1 punkt. Wielokrotne wywołanie powoduje przyjemny dla oka, płynny scrolling. Wywołanie od adresu **adrp**. Przykład wywołania (scrolling ekranu w górę o 8 linii, czyli o 1 wiersz):

FOR I=1 TO 8: RANDOMIZE USR 64Ø46 : NEXT I

Program 49

```
9620 REM Plynny scroll w gore
9621 DATA 64046,9621
9622 DATA "0100FF5104C578CDB110"
9623 DATA "22E506207E71121C2C76"
9624 DATA "10F9D1C178FEBF20E9D9"
9625 DATA "C9C9",""
```

4. **Lustrzane odbicie ekranu** (43 bajty, **adrp**=64Ø74)
Procedura powoduje poziome odwrócenie zawartości ekranu: punkty z lewego krańca ekranu zostaną zamienione z punktami prawego skraju itp. Instrukcją **POKE adrp+28, Ø** modyfikujemy procedurę tak, że tylko prawa połowa ekranu stanie się lustrzanym odbiciem lewej, lewa zaś pozostanie nie zmieniona. Odtworzenie stanu wyjściowego: **POKE adrp+28, 18**. Uruchamianie procedury od adresu **adrp**. Przykład zastosowania (dwukrotne odwrócenie ekranu):
RANDOMIZE USR 64Ø74 : PAUSE 5Ø : RANDOMIZE USR 64Ø74

Program 50

```
9630 REM Lustrzane odbicie
9631 DATA 64074,9631
9632 DATA "11004006D80E1021208E"
9633 DATA "0019E5C5060082B7CFE76"
9634 DATA "581A30101FCB1610FBBD"
9635 DATA "1F12130D20ECC1D110FF"
9636 DATA "E0C9467778181F1E7",""
```

5. **Wymienny ekran** (19 bajtów, **adrp**=64117)
Ekran mieści tylko 22—24 linie po 32 znaki. Często zachodzi jednak potrzeba szybkiego dostępu do większej ilości informacji na ekranie. Przykładem może być program użytkowy lub edukacyjny, przeznaczony dla niewprawnego jeszcze użytkownika komputera. W razie jakichkolwiek trudności użytkownik powinien mieć możliwość wywołania pojedynczym wciśnięciem klawisza odpowiedniego tekstu z objaśnieniami i przykładami poprawnych zleceń. Podobna sytuacja występuje także w grach, gdy zamiennie przełączamy między np. mapą terenu i pulpitem kierowcy pojazdu.
Poniższa procedura pozwala łatwo rozwiązać ten problem. Wystarczy zarezerwować w pamięci operacyjnej obszar długości 6912 bajtów, czyli wielkości pamięci ekranu. Procedura za każdym wywołaniem zamienia zawartość pamięci ekranu i pamięci pomocniczej. Można np. przygotować tekst objaśnień i wysłać je do pamięci pomocniczej, a następnie uruchomić właściwy program. W razie potrzeby ponownie wywołujemy procedurę. Na ekranie pojawia się blok objaśnień, poprzednia zaś zawartość ekranu w nie

zmienionym stanie przechowana jest w pamięci pomocniczej i wraca na ekran po kolejnym wywołaniu procedury. Pomocnicza pamięć ekranu zlokalizowana została pod adresem 56ØØØ. Trzeba to uwzględnić, wykonując przed ewentualnym użyciem procedury instrukcję **CLEAR** 55999. W razie potrzeby można przenieść pamięć pomocniczą pod inny adres. Adres pamięci pomocniczej zawarty jest w komórkach **adrp+1**, **adrp+2** i może być zmodyfikowany np. podprogramem **DPOKE**. Uwaga! Przy pierwszym użyciu procedury ekran może stać się czarny. Jest to normalne: na miejsce dotychczasowej pamięci ekranu wprowadzono zawartość wyzerowanego obszaru pamięci. Wystarczy skasować ekran zleceniem **CLS**. Przykład użycia (zapisanie obydwu ekranów i włączanie ich na zmianę):

```
10 CLS : PRINT "TO JEST TEST"
20 RANDOMIZE USR 64117: CLS
30 CIRCLE 100,70,50: CIRCLE 150,80,40
40 RANDOMIZE USR 64117: PAUSE 75
50 GO TO 40
```

Program 51

```
9640 REM Wymienny ekran
9641 DATA 64117,9641
9642 DATA "21C0DA1100401A4E77EB"
9643 DATA "79122313 7AFE5B20F4A8"
9644 DATA "C9C9","" 
```

6. **Dodatkowa pamięć ekranu** (39 bajtów, **adrp**=64136)
Tym razem we wspólnym bloku danych umieszczono procedury wykonujące trzy różne funkcje. Wszystkie one operują na pomocniczej pamięci ekranu. Procedura pierwsza (wywołanie od adresu **adrp**) kopiuje aktualną treść ekranu do pamięci pomocniczej, pozostawiając ekran w nie zmienionym stanie. Procedura druga działa odwrotnie: kopiuje pamięć pomocniczą do pamięci ekranu, nie zmieniając treści pamięci pomocniczej (wywołanie od **adrp+3**). Procedury mogą mieć zastosowanie np. przy tworzeniu kompozycji graficznych na ekranie. Po każdym udanym zabiegu możemy przechować ekran w pamięci pomocniczej i bez obawy o zniszczenie rysunku podejmować eksperymenty plastyczne. W razie niepowodzenia możemy odtworzyć pierwotny wygląd ekranu wywołaniem drugiej procedury kopiującej. Kopiowanie obejmuje także atrybuty.
Procedura trzecia umożliwia stopienie treści ekranu i pamięci pomocniczej. Jest to przydatne np. w przypadku, gdy opracowywany rysunek zawiera stałe elementy, które można przechowywać w pamięci pomocniczej i w razie potrzeby wrysowywać w ekran. Atrybuty nie są uwzględnione przy stapianiu.

Kolorystyka ekranu zostanie więc zachowana. Wywołanie od adresu **adrp+19**.
Wszystkie trzy procedury zakładają, że pamięć pomocnicza zaczyna się od komórki 56ØØØ. Można to łatwo zmienić (patrz: wymienny ekran). Adres pamięci pomocniczej zapisany jest w komórkach **adrp+8**, **adrp+9** dla procedur kopiujących i w komórkach **adrp+2Ø**, **adrp+21** dla procedury stapiającej ekrany. Przykład użycia (tworzony rysunek jest przechowywany w pamięci, tworzony jest nowy rysunek i odbywa się stapianie):

```
1Ø CLS : CIRCLE 11Ø, 7Ø, 6Ø : CIRCLE 9Ø, 9Ø, 17
2Ø RANDOMIZE USR 64136 : CLS
3Ø PRINT "DRUGI RYSUNEK": CIRCLE 1ØØ, 8Ø, 73
4Ø PAUSE 5Ø : RANDOMIZE USR 64155
```

Program 52

```
9650 REM Dodatkowa Pamiec ekranu
9651 DATA 64136,9651
9652 DATA "AF18013711C0DA2100CB"
9653 DATA "403001EB01001BEDB015"
9654 DATA "C911C0DA21004001 18EE"
9655 DATA "001AB6771323 10F90D93"
9656 DATA "20F6C9DF",""
```

7. **Inwersja ekranu** (16 bajtów, **adrp**=64175)
Procedura dokonuje graficznej inwersji ekranu: wszystkie punkty zapalone stają się punktami tła, wszystkie punkty tła zostają zapalone. Wywołanie od adresu **adrp**. Przykład użycia (ekran na 2 sekundy staje się negatywem):

```
RANDOMIZE USR 64175: PAUSE 1ØØ : RANDOMIZE USR 64175
```

Program 53

```
9660 REM Inwersja ekranu
9661 DATA 64175,9661
9662 DATA "21004001 18007E2F779E"
9663 DATA "2310FA0D20F7C91A",""
```

8. **Rysowanie znaków w dowolnym powiększeniu** (9Ø bajtów, **adrp**=64191)
Ostatnia procedura pozwala na rysowanie znaków w dowolnym miejscu ekranu (współrzędne ustalane są z dokładnością do 1 punktu) i w dowolnym powiększeniu, ustalanym oddzielnie dla osi X i Y. W odróżnieniu od procedur poprzednich tutaj wymagane jest przekazanie parametrów: kodu znaku, współrzędnych lewego górnego rogu jego pola oraz powiększenie dla obu

osi. Przekazywanie parametrów odbywa się instrukcjami **POKE**. Kod znaku może być liczbą z zakresu 32—127 (główny generator znaków) lub 144—164 (UDG), przy czym położenie obydwu obszarów wzorców jest obojętne (wzorce mogą być przemieszczone). Kod znaku należy wpisać do komórki **adrp+1**, współrzędną X do **adrp+3**, Y do **adrp+4**, powiększenie w osi X do **adrp+6**, a powiększenie w osi Y do **adrp+7**. Jeśli jakieś parametry, np. powiększenie lub jedna ze współrzędnych, nie zmieniają się w kolejnych wywołaniach, to oczywiście nie trzeba ich modyfikować. Wywołanie od adresu **adrp**. Przykład zastosowania (wyrysowanie liter A, B, C, D w środku ekranu. Powiększenie w osi X wynosi 3, w osi Y — 6):

```
10 POKE 64197,3: POKE 64198,6: POKE 64195,90
20 FOR C=1 TO 4: POKE 64192,64+C
30 POKE 64194,50+30*C: RANDOMIZE USR 64191
40 NEXT C
```

Program 54

```
9670 REM Rysowanie znakow
9671 DATA 64191,9671
9672 DATA "3E4101644621030SE538"
9673 DATA "ED5B365CCB7F2806ED3F"
9674 DATA "5B7B5CD6906F26002956"
9675 DATA "292919E5DDE12A8D5C21"
9676 DATA "228F5CE1C5D90608D973"
9677 DATA "54DD7E00D90E08D95DD4"
9678 DATA "87F5C5D5E5DCE922E1C3"
9679 DATA "D1C1F10C1D20F1D90DA3"
9680 DATA "D920EAC105C51520DD80"
9681 DATA "DD23D905D920D5C1C936"
9682 DATA ""
```

Dla szybkiego załadowania do pamięci wszystkich ośmiu procedur wystarczy zlecenie:

RESTORE 96ØØ:FOR I = 1 TO 8:GO SUB 999Ø:NEXT I

Zaprezentowane programy maszynowe, mimo że bardzo proste i krótkie, realizują zadania niemożliwe do rozwiązania w języku BASIC z uwagi na niewystarczającą prędkość pracy. Niech będzie to zachętą do bliższego zapoznania się z językiem maszynowym: narzędziem dającym programiście praktycznie nieograniczoną władzę nad mikrokomputerem.

"Życie prywatne" interpretera

Organizacja pamięci w ZX Spectrum nie jest już zagadką. Radzimy sobie z bitami i bajtami, ładujemy i uruchamiamy programy maszynowe. Najwyższy czas, by przyjrzeć się bliżej pracy interpretera języka BASIC.
Wprowadziliśmy z klawiatury wiersz zawierający instrukcję i zakończyliśmy go klawiszem ENTER. W tym momencie podjął pracę interpreter, analizując wiersz i badając jego zgodność z regułami języka. W przypadku wykrycia błędnej składni nastąpi sygnalizacja tego faktu i wiersz nie zostanie przyjęty. Jeśli wiersz był poprawny i poprzedzony numerem, zostanie on po prostu wpisany do pamięci programu. W przeciwnym razie interpreter przystąpi od razu do wykonywania zawartych w wierszu instrukcji.
Wykonując program, interpreter analizuje bajt po bajcie (znak po znaku) zapisany w pamięci program — tak, jakbyśmy sami odczytywali go z ekranu. Na początku każdej instrukcji oczekiwany jest rozkaz. Z jego rozpoznaniem nie ma kłopotu. Każde słowo kluczowe (**LET**, **GO TO**, **INT** itp.) kodowane jest w pamięci za pomocą pojedynczego bajtu o wartości większej od 164. Takich kodów nie mają żadne inne znaki. Mimo że na ekranie słowo kluczowe jest ciągiem kilku znaków, dla maszyny jest zawsze pojedynczym bajtem. Przekonajmy się o tym, pisząc zlecenie:

PRINT CHR$ 249

Mimo że wyprowadzamy pojedynczy znak, na ekranie pojawiło się całe słowo **RANDOMIZE**. Liczba 249 jest właśnie jego wewnętrznym kodem, podobnie jak np. 251 jest kodem **CLS**, **245** — **PRINT** lub 241 — **LET**. Przy wykonywaniu instrukcji **PRINT** rozpoznany kod 249 został przekształcony na pełne słowo. Stało się to jednak tylko na użytek operacji wyjścia (wyświetlania). Podobnie będzie w operacji wejścia. Napiszemy zlecenie:

INPUT A$: PRINT LEN A$

w odpowiedzi na znak zachęty wciśnijmy SYMBOL SHIFT, a potem klawisz "G". Pojawi się słowo **THEN**. Po wciśnięciu ENTER wyświetlona zostanie długość łańcucha **A$**. Wyniesie ona tylko 1. Napiszemy:

PRINT A$: CODE A$

Przedstawianie słów kluczowych w zakodowanej postaci ma ważne zalety. Pamięć programu wykorzystywana jest oszczędniej. Sam program wykonywany jest szybciej. Kiedy podczas analizy programu interpreter napotka bajt o wartości przewyższającej 164, od razu będzie wiadomo, że symbolizuje on rozkaz lub nazwę funkcji. W takim przypadku interpreter wykona po prostu odpowiednią procedurę maszynową, odpowiedzialną za wykonanie konkretnej instrukcji lub obliczenie określonej funkcji. Jeśli będzie to np. **DATA**, odpowiedni program maszynowy spowoduje po prostu jej "przeskoczenie", po czym nastąpi powrót do głównego programu, analizującego i wykonują-

cego kolejne instrukcje języka BASIC (tzw. główna pętla interpretera). Po wykryciu instrukcji **CLS** uruchomiony zostanie program maszynowy, zapisujący pamięć obrazu i atrybutów stałymi wartościami — podobnie jak czynił to nasz program nr 42 lub jego maszynowy odpowiednik.

W języku BASIC posługujemy się nie adresami komórek pamięci, lecz symbolicznymi nazwami zmiennych. Wartość zmiennej zapamiętana jest jednak w konkretnych komórkach pamięci. Jeśli podczas np. obliczania wartości wyrażenia interpreter natknie się na zmienną, rozpocznie przeszukiwanie tzw. tablicy zmiennych (słownika zmiennych). W tablicy tej umieszczone są kolejno: nazwa zmiennej wraz z informacjami o jej charakterze (zmienna prosta, tablica, liczba, łańcuch) oraz wartość zmiennej (w przypadku tablic: zbiór wartości). Każda zmienna zadeklarowana instrukcją **DIM** lub operacją przypisania w instrukcjach **LET** lub **FOR** jest dopisywana na końcu tablicy zmiennych. Adres tablicy zmiennych zawarty jest w dwubajtowej zmiennej systemowej **VARS** (adres: 23627). Przyjrzymy się budowie tej tablicy. Najpierw jednak oczyścimy pamięć zleceniem **NEW**. Teraz w trybie bezpośrednim wprowadzimy kilka zmiennych:

```
LET a=5
LET b$="ABC"
DIM x(2)
LET x(1) = 78
LET x(2) = 259
```

Następnie ustalimy, gdzie w pamięci zaczyna się tablica zmiennych:

```
PRINT PEEK 23627 +256*PEEK 23628
```

Ponieważ w pamięci nie ma żadnego programu, powinniśmy uzyskać wynik 23755. Wyświetlmy na ekranie zawartość tablicy. Posłużmy się w tym celu pętlą **FOR... NEXT**, którą — ważne! — także trzeba uruchomić w trybie bezpośrednim:

```
FOR i=23755 TO 23782: PRINT i, PEEK i: NEXT i
```

Otrzymamy na ekranie kolumnę adresów i odpowiadających im zawartości komórek pamięci. Pierwszych sześć wierszy wyglądać będzie tak:

```
23755   97
23756   Ø
23757   Ø
23758   5
23759   Ø
2376Ø   Ø
```

Liczba 97 ma w przedstawieniu dwójkowym postać: $Ø11ØØØØ1_2$ i zawiera informację zarówno o typie, jak i o nazwie zmiennej. Trzy najstarsze bity kodują typ. Kombinacja Ø11 oznacza: zmienna liczbowa prosta o nazwie

jednoliterowej. Pięć bitów mniej znaczących koduje jednoliterową nazwę. Wiadomo, że nazwa taka musi być literą. Wszystkie małe litery mają identyczne trzy bity najstarsze: Ø11. Używamy wprawdzie także i zmiennych nazwanych dużymi literami, Spectrum jednak, jak pamiętamy, liter małych i dużych w nazwach nie rozróżnia. Wiemy teraz, dlaczego. Wszystkie nazwy kodowane są wyłącznie małymi literami. Dla równoczesnego zapisania liter małych i dużych pięć bitów nie wystarcza. Jeśli do pięciu bitów młodszych (ØØØØ1) dopiszemy trzy starsze (Ø11) bity kodu małych liter, otrzymamy ciąg $Ø11ØØØØ1_2 = 97_{1Ø}$). Odpowiada to małej literze "a". Tak nazwaliśmy pierwszą zmienną. Reguły kodowania wieloliterowych nazw zmiennych są nieco bardziej złożone. Ostatnia litera nazwy ma najstarszy bit ustawiony na 1. Zmienne liczbowe o nazwach wieloznakowych mają oddzielny kod typu: 1Ø1. Pięć bitów następujących po nazwie przeznaczonych jest na wartość zmiennej. Spectrum stosuje dwa sposoby zapisu liczb. Liczby całkowite o wartości bezwzględnej większej od 65535 oraz wszystkie liczby ułamkowe zapisywane są w tzw. formacie wykładniczym, z wykorzystaniem wszystkich pięciu bajtów. Liczby całkowite nie większe od 65535 zapisywane są w znanym nam już formacie dwubajtowym, wykorzystującym tylko trzeci i czwarty bajt pola wartości. Młodszy bajt zapisywany jest w bajcie trzecim, bajt starszy — w czwartym.

Zmiennej **a** przypisaliśmy wartość 5. Została ona zapamiętana w formacie liczb całkowitych. Zapis małych liczb całkowitych w tym formacie nieco przyspiesza niektóre obliczenia.

Czas przyjrzeć się kolejnej zmiennej:

23761	**66**
23762	**3**
23763	**Ø**
23764	**65**
23765	**66**
23766	**67**

Pierwszy bajt $66_{1Ø}$ = ØØ1ØØØ1Ø zawiera typ (ØØ1 — zmienna łańcuchowa prosta) i nazwę (ØØØØ1). Tu także obowiązują reguły kodowania znane z poprzedniego przykładu. Po uzupełnieniu otrzymamy $Ø11ØØØ1Ø_2$ = $98_{1Ø}$, co odpowiada literze "b". Następne dwa bajty przedstawiają długość zmiennej łańcuchowej (w naszym przykładzie: 3). Kolejne bajty to już właściwa zawartość łańcucha. Liczby 65, 66 i 67 to kody liter "A", "B" i "C", z których aktualnie złożony jest łańcuch **b$**.

Przejdźmy do trzeciej zmiennej:

23767	**152**
23768	**13**
23769	**Ø**
2377Ø	**1**

23771	2
23772	Ø
23773	Ø
23774	Ø
23775	78
23776	Ø
23777	Ø
23778	Ø
23779	Ø
23780	3
23781	1
23782	Ø

Bajt $152_{10} = 10011000_2$. Kod 100_2 oznacza "tablica liczbowa". 11000_2 uzupełnione do postaci 01111000_2 przedstawia literę "x". Następne dwa bajty to ogólna długość obszaru zajętego przez pozostałe elementy tablicy (13). Czwarty bajt przedstawia liczbę wymiarów. Nasza tablica jest tylko jednowymiarowa. Kolejne dwa bajty określają maksymalną wartość indeksu dla pierwszego (w naszym przypadku: jedynego) wymiaru. Zadeklarowaliśmy tylko dwa elementy. Dalej następują już, jedno po drugim, pięciobajtowe pola wartości poszczególnych elementów tablicy.
Wartości obu elementów tablicy (78 i 259=3+256•1) także zapisane zostały w formacie liczb całkowitych.
Zawartość tablicy symboli możemy modyfikować instrukcją **POKE**. Użyjmy zlecenia:

POKE 23758, 44: PRINT a

Na ekranie pojawi się liczba 44, mimo że nie przeprowadziliśmy względem zmiennej **a** takiego przypisania! Podobnie postąpimy teraz ze zmienną tekstową **b$**:

POKE 23765, 108 : POKE 23766, 97 : PRINT b$

Zamiast "ABC" na ekranie pojawi się "Ala"!
Sztuczki z modyfikacją zmiennych instrukcją **POKE** wymagają dużej ostrożności. Położenie zmiennych w pamięci ulega częstym zmianom.
Tablica symboli umieszczona jest zawsze bezpośrednio za programem. Dodanie lub usunięcie linii programu zmienia jego objętość, a więc i położenie końca. Każda taka operacja powoduje zatem przesunięcie tablicy zmiennych. Ale to nie wszystko. Przy próbie przypisania np. zmiennej **b$** z naszego przykładu nowej wartości, zostałaby ona usunięta z zajmowanego miejsca i przeniesiona na sam koniec tablicy symboli, pozostałe zaś zmienne przesunęłyby się w stronę początku tablicy i pokryły obszar zajmowany uprzednio przez **b$**.
Interpreter przeszukuje tablicę zmiennych zawsze od początku do końca. W dużym programie występuje zazwyczaj co najmniej kilkadziesiąt różnych

zmiennych. Poszukiwanie zmiennej znajdującej się na końcu tablicy może wtedy nieco potrwać. To jeszcze jedna przyczyna dość wolnej pracy programów w języku BASIC. Wygodę posługiwania się symbolicznymi nazwami zmiennych w trybie bezpośrednim trzeba czymś okupić...
Użyjmy ponownie zlecenia **NEW**. Przyjrzyjmy się teraz budowie właściwego programu, a właściwie jego reprezentacji w pamięci operacyjnej. Adres pierwszego bajtu programu umieszczony jest w dwubajtowej zmiennej systemowej **PROG** (adres: 23635). W normalnych warunkach zawartość tej zmiennej wynosi 23755 (gdy nie ma programu, np. po **NEW** — 23754). Wprowadźmy króciutki program:

```
10 CLS
20 PRINT "B"
```

Aby prześledzić sposób zakodowania programu w pamięci, użyjemy znowu pętli uruchomionej w trybie konwersacyjnym:

```
FOR a=23755 TO 2377Ø: PRINT a, PEEK a: NEXT a
```

Oto ciąg liczb, który ujrzymy na ekranie w następstwie wykonania pętli:

23755	Ø
23756	1Ø
23757	2
23758	Ø
23759	251
2376Ø	13
23761	Ø
23762	2Ø
23763	5
23764	Ø
23765	245
23766	34
23767	66
23768	34
23769	13
2377Ø	225

Każdy wiersz programu rozpoczyna się czterobajtowym nagłówkiem. Dwa pierwsze bajty nagłówka przechowują numer wiersza. Ale uwaga! Numer ten — zupełnie wyjątkowo — zapisany jest w odwrotnej kolejności: najpierw bajt starszy, potem młodszy. Dlaczego? Wiemy, że numer wiersza w Spectrum ograniczony jest do 9999.9999_{10} = ØØ1ØØ111 ØØØØ1111$_2$. Widzimy, że dwa najstarsze bity bardziej znaczącego bajtu numeru wiersza będą zawsze zerowe, podczas gdy bajt mniej znaczący przyjmuje dowolne wartości z zakresu Ø—255, a co za tym idzie, dowolne kombinacje bitów. Tę właściwość starszego bajtu wykorzystuje się do sygnalizacji końca progra-

mu. Jeśli przy próbie odczytania nagłówka uzyska się starszy bajt z ustawionym co najmniej jednym spośród dwóch najstarszych bitów, będzie to znaczyło, że nie jest to już linia programu i że dalszych wierszy już nie ma. Aby móc użyć jako wskaźnika pojedynczego bajtu, należało bajt starszy zapisać w pierwszej kolejności. Trzeci i czwarty bajt nagłówka zawierają długość pozostałej części wiersza. Właściwa treść wiersza zaczyna się od piątego bajtu i kończy symbolem ENTER (kod: 13). Treścią wiersza są instrukcje, separatory itp.

Spójrzmy znów na ekran: numer pierwszego wiersza wynosił 1Ø. Zgadza się. Długość wiersza = 2 + Ø*256 = 2. Rzeczywiście, wiersz składa się tylko z dwóch bajtów: rozkazu **CLS** (kod: 251) i znaku ENTER (kod: 13). Bezpośrednio za ENTER następuje nagłówek kolejnego wiersza. Zawiera on numer wiersza 2Ø i długość wiersza 5. Na treść wiersza składają się tym razem: kod rozkazu **PRINT** (245), kod cudzysłowu (34), kod litery "B" (66), kod drugiego cudzysłowu i ENTER. Umieszczony tuż za ENTER bajt o wartości 225 wskazuje, że tutaj kończy się program. Normalnie tuż za tekstem programu znajduje się tablica symboli. Pierwszy bajt pierwszego elementu tablicy symboli spełnia równocześnie funkcję wskaźnika końca programu. Tak jest i tym razem. $225_{10}=111ØØØØ1_2$. Kod typu 111_2 oznacza zmienną sterującą pętli **FOR... NEXT**, a kod $ØØØØ1_2$, uzupełniony do $Ø11ØØØØ1_2$, reprezentuje literę "a". Zmienna "a" co prawda nie wystąpiła w naszym programie, ale użyliśmy jej w pętli wyświetlającej kolejne komórki pamięci.

Rozsądnie używając instrukcji **POKE** możemy oddziaływać także i na zapamiętany program. Wprowadźmy zlecenie:

POKE 23759, 249 : POKE 23767, 49

Po wylistowaniu programu od razu zauważymy zmiany:

```
1Ø RANDOMIZE
2Ø PRINT "1"
```

Spróbujmy zmienić numer drugiego wiersza programu:

POKE 23762, 2

Rzecz niecodzienna: wiersz o niższym numerze znalazł się w programie za wierszem o numerze wyższym:

```
1Ø RANDOMIZE
2  PRINT "1"
```

Wprowadzenie linii o numerze zerowym jest normalnie niemożliwe. Wyzerujmy więc numer pierwszej linii zleceniem:

POKE 23756, Ø

Pierwsza linia ma od tej pory numer Ø, i co najważniejsze, nie pozwala się

sprowadzić do edycji ani skasować mniej brutalnymi środkami niż zlecenie **NEW!** Aby odtworzyć numer wiersza, wystarczy wpisać poprzednią wartość młodszego bajtu:

POKE 23756, 1Ø

Przekonaliśmy się, że program w języku BASIC jest najnormalniejszym ciągiem bajtów, na który możemy wpływać na równi z interpreterem. Nie święci garnki lepią. Potrzebna jest tylko wiedza.

Niewidoczne, lecz jakże istotne

Przedstawiony w poprzednim rozdziale programik przykładowy miał miłą cechę: zawartość pamięci wiernie odpowiadała obrazowi na ekranie (nie licząc oczywiście bajtów organizacyjnych: nagłówka i ENTER). W programie występują jednak elementy nie zawsze widoczne. Wprowadźmy **NEW** i zapiszmy instrukcję:

1Ø LET x=5

po czym odpowiednią pętlą zbadajmy jej reprezentację w pamięci:

FOR a=23755 TO 23769: PRINT a, PEEK a: NEXT a

Oto, co otrzymamy:

23755	**Ø**	
23756	**1Ø**	
23757	**11**	
23758	**Ø**	
23759	**241**	(LET)
2376Ø	**12Ø**	(x)
23761	**61**	(=)
23762	**53**	(5)
23763	**14**	
23764	**Ø**	
23765	**Ø**	
23766	**5**	
23767	**Ø**	
23768	**Ø**	
23769	**13**	

Obrazowi linii programu na ekranie odpowiadają bajty 23759—23762. Bajty 23763—23768, obecne w programie, nie mają odzwierciedlenia na ekranie, są utajnione. Co przedstawiają?
Ostatnim elementem wprowadzonej instrukcji była stała liczbowa 5. Podczas wstępnej analizy, przed przesłaniem wiersza do pamięci, interpreter prze-

kształcił symboliczną postać liczby (cyfra "5") na wewnętrzną, typową, pięciobajtową reprezentację w standardowym formacie (tutaj: całkowitym) i dopisał ją bezpośrednio za postacią symboliczną, dodając na początku znak ostrzegawczy w postaci bajtu o wartości 14_{10}. Bajt taki w trakcie wykonywania lub listowania programu informuje interpreter, że następnych 5 bajtów to stała w formacie wewnętrznym. Po co przekształcać liczbę na postać wewnętrzną już w trakcie wprowadzania do pamięci? Czy nie wystarczyłoby tego dokonać dopiero w trakcie realizacji programu? Owszem, to możliwe. Postępuje tak wiele innych interpreterów. Przekształcanie liczby z postaci dziesiętnej na binarną jest jednak zazwyczaj zabiegiem czasochłonnym, wymagającym wykonania np. wielu mnożeń. Jeśli stała jest fragmentem np. wyrażenia obliczanego w pętli programowej, to przekształcanie jej na postać wewnętrzną odbywałoby się niepotrzebnie setki i tysiące razy, przy każdym powtórzeniu pętli.
Przyjęte w Spęctrum rozwiązanie kosztem powiększenia objętości programu skraca czas jego wykonania. Pamiętajmy, że w fazie realizacji wykorzystywana jest wyłącznie postać binarna stałej, postać symboliczna zaś to jedynie dekoracja, potrzebna przy listingu i edycji wiersza. Może to mieć zabawne konsekwencje. Wprowadźmy taki oto jednowierszowy programik:

```
10 PRINT 127
```

Jego reprezentacja w pamięci (bez nagłówka) będzie następująca:

23759	**245**	(PRINT)
23760	**49**	(1)
23761	**50**	(2)
23762	**55**	(7)
23763	**14**	
23764	**0**	
23765	**0**	
23766	**127**	
23767	**0**	
23768	**0**	
23769	**13**	

Po uruchomieniu programiku zleceniem **RUN** posłuszny komputer wyświetli "127". Teraz zleceniem:

```
POKE 23766, 9
```

zmienimy wartość stałej w postaci dwójkowej. Wykonajmy ponownie **LIST** i **RUN**. Obraz wylistowanej linii nie zmienił się. W wyniku wykonania programu komputer wyświetlił jednak nie 127, lecz 9! Opisana sztuczka może pomóc w ochronie programu przed "rozgryzieniem" go przez osoby niepowołane. Listing programu nie odpowiada jego rzeczywistej, wewnętrznej treści. Osobom stosującym podobne sztuczki zaleca się jednak ich staranne

udokumentowanie na własny użytek. W przeciwnym razie wprowadzanie przez autora jakichkolwiek poprawek w programie stać się może niezłą łamigłówką — zwłaszcza gdy od zmyślnego "zaPOKEowania" programu minęły ponad dwa tygodnie!
Utrudnieniem życia dla nie upoważnionych amatorów cudzych programów może być zmiana wskaźnika długości wiersza (trzeci i czwarty bajt nagłówka). W trakcie normalnej realizacji programu lub jego listowania nie jest on wykorzystywany. Dopiero przy wykonywaniu skoku do zadanego wiersza (np. **GO TO**) komputer przeszukuje program, poczynając od pierwszego wiersza. Gdy po zbadaniu nagłówka okaże się, że aktualny numer wiersza jest mniejszy od szukanego, interpreter przeskakuje od razu cały wiersz. Przeskok następuje o liczbę bajtów podaną w trzecim i czwartym bajcie nagłówka. Gdybyśmy jednak zmienili wskaźnik długości, komputer za jednym razem mógłby przeskoczyć więcej niż 1 wiersz. Oto przykład:

Program 55

```
10 LET X=2
20 LET X=X*2
30 PRINT X
40 GO TO 20
```

Po uruchomieniu program wyświetla kolejne potęgi liczby 2. W znany sposób: pętlą

FOR a=23755 TO 23807: PRINT a, PEEK a: NEXT a

przeanalizujmy wewnętrzną postać programu. Ostatnim bajtem wiersza nr 20 jest bajt 23786. Wskaźnik długości wiersza nr 10 wynosi 11 (bajty od 23759 do 23769). Zmieńmy go tak, by wskazywał nie na początek wiersza 20, lecz 30:

POKE 23757, 22

Listowanie programu nie ujawni żadnych zmian. Dopiero po **RUN** będzie niespodzianka: na ekranie pojawi się kolumna samych dwójek. Wytłumaczenie jest proste. Wykonując **GO TO 20** komputer zacznie szukać wiersza 20. Pierwszy wiersz ma numer mniejszy od zadanego, komputer zechce więc go przeskoczyć. Ponieważ jednak wskaźnik pokazuje na początek wiersza 30, wiersz 20 nie zostanie w ogóle wzięty pod uwagę. Skok nastąpi więc do wiersza 30, choć bynajmniej nie wynika to z listingu.

Ukryte znaki

W programie występować może jeszcze więcej niewidocznych, lecz istotnych elementów. Należą do nich wszystkie znaki sterujące postacią wyświet-

lanej informacji: kolorem tła i wypełnienia, jaskrawością, migotaniem itd., znaki tabulacji i pozycjonowania kursora. Okazuje się, że **AT**, **TAB**, **INK** itd. nie wpływają bezpośrednio na zawartość zmiennych systemowych, odpowiedzialnych za postać wyświetlanej informacji. Wysyłają one sekwencje (ciągi) znaków sterujących, które włączone są w strumień innych znaków kierowanych na ekran w instrukcji np. **PRINT**. Wyprowadzaniem informacji znakowej na ekran zarządza specjalna procedura maszynowa, stanowiąca element pamięci ROM. Przechodzą przez nią wszystkie dyspozycje wpływające na postać ekranu. Jeśli istnieją specjalne znaki sterujące, to będziemy w stanie obejść się bez nie zawsze wygodnych i ekonomicznych modyfikatorów. Możemy znaki sterujące wmontować bezpośrednio w wyprowadzane ciągi symboli. Zacznijmy od systematycznego poznawania sterujących sekwencji:

AT	kod	22
TAB	kod	23

Za każdym z tych znaków następować muszą 2 bajty-argumenty (jako argumenty potraktowane zostaną 2 następne, wysłane po nich znaki). Pierwszy bajt określa po **AT** współrzędną pionową, drugi — poziomą. Po **TAB** pierwszy bajt określa współrzędną poziomą, drugi zaś jest bez znaczenia. Kolejna grupa znaków sterujących wymaga tylko pojedynczego bajtu jako argumentu:

INK	kod	16
PAPER	kod	17
FLASH	kod	18
BRIGHT	kod	19
INVERSE	kod	2Ø
OVER	kod	21

Wartość argumentu musi należeć do zbioru wartości dopuszczalnych dla danego modyfikatora. W przeciwnym razie nastąpi sygnalizacja błędu. Znaki sterujące można włączyć w budowane programowo łańcuchy za pomocą funkcji **CHR$**:

```
1Ø LET a$=CHR$ 22+CHR$ 1Ø+CHR$ 12+"ABCD"+CHR$ 2Ø+CHR$ 1+"CDEF"
2Ø PRINT a$
```

Po uruchomieniu programu napis wyświetli się pośrodku ekranu: w wierszu 1Ø poczynając od pola 12. Powoduje to **AT** (kod 22) wraz z argumentami. Konieczność wprowadzenia ich w łańcuch a$ za pośrednictwem funkcji **CHR$** wynika stąd, że bajty te traktowane są formalnie jak znaki (innych elementów w zmiennej tekstowej być przecież nie może). Pierwsze cztery znaki zostaną ukazane normalnie, następne cztery — inwersyjnie. Sprawi to przełącznik **INVERSE** (kod 2Ø) wraz z argumentem 1.

Znaki sterujące wprowadzić można także w treść programu. Dostępne z klawiatury modyfikatory pozwalają wpływać jednak tylko na barwę tła znaków i inwersję. Chcąc otrzymać ciekawsze efekty, trzeba będzie sięgnąć znowu do instrukcji **POKE**. Najpierw wprowadźmy zlecenie **NEW**, a po nim — następującą instrukcję:

```
1Ø PRINT "TEST MODYFIKATOROW"
```

Pomiędzy cyfrą Ø a rozkazem **PRINT** należy wcisnąć klawisz spacji dokładnie cztery razy. Teraz wykonajmy zlecenia:

```
POKE 23759, 17
POKE 2376Ø, 5
```

Listując program zauważymy zniknięcie dwóch spacji oraz błękitną barwę tła znaków w wierszu. Pierwsze dwie spacje zastąpiliśmy bajtem 17 (**PAPER**) oraz 5 (wskaźnik koloru błękitnego). Wykonajmy następne dwa zlecenia:

```
POKE 23761, 16
POKE 23762, 5
```

Te dwa bajty wprowadziliśmy w miejsce pozostałych dwóch spacji. Oznaczają one modyfikację koloru atramentu (16=**INK**) na błękitny. Po **LIST** na ekranie pojawi się tylko błękitny pasek: tak właśnie wyglądają błękitne litery na błękitnym tle... Gdyby w obu modyfikatorach użyć argumentu nie 5, lecz 7 (kolor biały), w listingu ukazałby się jedynie numer wiersza. Uruchomiony zleceniem **RUN** program działałby jednak nadal. Modyfikatory w treści programu są w trakcie wykonywania ignorowane — z wyjątkiem tych, które występują w stałych łańcuchowych. Powyższą sztuczkę łatwo wykryć, poprzedzając **LIST** zleceniem np.

```
PAPER 3: CLS
```

Wyświetlający się po numerze wiersza biały pas na ciemnym ekranie zdradza obecność utajnionej treści. Ciekawsze wyniki można uzyskać za pomocą znaku sterującego o kodzie 8. Tu nie ma już żadnych argumentów. Nie są potrzebne. Znak ten (**BS**, ang. backspace) powoduje cofnięcie pozycji wydruku o jedno pole w lewo. Wpiszmy teraz w miejsce pierwotnych spacji cztery znaki "kursor w lewo" (**BS**):

```
POKE 23759, 8 : POKE 2376Ø, 8
POKE 23761, 8 : POKE 23762, 8
```

Gdy wylistujemy program, na ekranie zobaczymy napis:

```
PRINT "TEST MODYFIKATOROW"
```

Doszło do tego w prosty sposób. Numer wiersza w trakcie listowania został jednak wyprowadzony. Gdy jednak komputer zaczął wyprowadzać kolejne znaki wiersza, 4 symbole **BS** przesunęły pozycję wydruku o 4 miejsca w lewo:

akurat do początku wiersza. Wyprowadzony jako następny symbol **PRINT** wyświetlił się więc w miejscu numeru wiersza. Odbyło się to bardzo szybko, nasz wzrok niczego nie zarejestrował. Znaków **BS** można oczywiście użyć do ochrony programu, ukrywając w nim istotne instrukcje.
Oto przykład: skasujmy stary program i wprowadźmy instrukcje:

```
1Ø INPUT X: LET X=X+Ø:5:          LET Y=X*2.11
2Ø PRINT Y
```

Pomiędzy dwukropkiem a słowem **LET** wciśnijmy spację 13 razy. Następnie pętlą:

```
FOR I=23776 TO 23788: POKE I, 8: NEXT I
```

zastąpimy 13 wprowadzonych uprzednio spacji symbolami **BS**. Po zleceniu **LIST** skutek będzie następujący:

```
1Ø INPUT X: LET Y=X*2.11
2Ø PRINT Y
```

Instrukcja **LET X=X+Ø.5** zawarta jest nadal w linii 1Ø. Bezpośrednio po jej wylistowaniu 13 kolejnych symboli **BS** cofa wskaźnik pozycji wydruku o 13 pozycji: akurat tyle, ile wynosi długość tej instrukcji na ekranie. Następna instrukcja zostanie więc wyświetlona na jej miejscu. Sztuczka wyjdzie na jaw dopiero po skierowaniu listingu na drukarkę lub sprowadzeniu wiersza do edycji. Zarówno wprowadzanie zmian w zapamiętanym już programie, jak też niekiedy i ich wykrywanie i usuwanie wymaga narzędzi do analizy programu. Narzędziem takim jest poniższy podprogram, wyznaczający adres wiersza o zadanym numerze:

Program 56

```
9850 REM SZUKAJWIERSZ(NW,AW)
9851 LET AW=FN D(23635)
9852 LET NNW=256*PEEK AW+PEEK (AW+1)
9853 IF NNW>9999 THEN LET AW=0: RETURN
9854 IF NNW=NW THEN RETURN
9855 LET AW=AW+4+FN D(AW+2)
9856 GO TO 9852
```

Przed wywołaniem podprogramu nadajemy zmiennej **NW** numer wiersza, którego adresu szukamy. Po powrocie zmienna **AW** wskazuje na pierwszy bajt nagłówka szukanego wiersza. Jeśli **AW=Ø**, to wiersz o zadanym numerze nie został odnaleziony.
Innym narzędziem, przydatnym przy analizie programów, będzie podprogram listujący zapamiętany wiersz w przejrzystej formie. Słowa kluczowe zostaną uzupełnione odpowiednim tekstem, a wartości stałych liczbowych będą zanalizowane i zostanie wyświetlona ich rzeczywista wartość:

Program 57

```
9830 REM LISTUJWIERSZ(AW)
9831 LET NW=256*PEEK AW+PEEK (AW+1)
9832 PRINT INVERSE 1;"WIERSZ ";NW,"ADRES:";AW
9833 LET AW=AW+4
9834 LET BW=PEEK AW: PRINT AW;"/";BW;TAB 10;: LET AW=AW+1
9835 IF BW>31 THEN PRINT CHR$ BW: GO TO 9834
9836 IF BW=14 THEN GO TO 9845
9837 IF BW=22 OR BW=23 THEN PRINT BRIGHT 1;PEEK (AW);" ";PEEK (AW+1);TAB 24;CHR$
 (BW+150): LET AW=AW+2: GO TO 9834
9839 IF BW>15 AND BW<22 THEN PRINT BRIGHT 1;PEEK (AW);TAB  24;CHR$ (BW+201): LET
 AW=AW+1: GO TO 9834
9843 IF BW=13 THEN PRINT BRIGHT 1;"ENTER": RETURN
9844 PRINT : GO TO 9834
9845 LET BW=FN D(23629): LET ABW=BW+1
9846 FOR Q=0 TO 4: POKE ABW+Q,PEEK (AW+Q): PRINT BRIGHT 1;PEEK (AW+Q);" ";: NEXT
 Q
9847 PRINT " #";BW
9848 LET AW=AW+5: GO TO 9834
```

Podprogram wymaga, aby zmienna **AW** zawierała adres pierwszego bajtu nagłówka wiersza. Można tu doskonale wykorzystać podprogram **SZUKAJ-WIERSZ** (985Ø). Nagłówek, zawierający numer i adres początkowy wiersza, wyprowadzany jest inwersyjnie. Po nim następują kolejne bajty poprzedzone ich adresami. W razie napotkania modyfikatora jego argumenty zostaną umieszczone z powiększoną jaskrawością w tym samym wierszu, co kod. Podobnie stałe w formacie binarnym: 5 kolejnych bajtów umieszczonych jest w polu o podwyższonej jaskrawości w tym samym wierszu, co symbol 14. Po bajtach wyświetlona jest ich wartość, poprzedzona krzyżykiem (hash). Także ENTER sygnalizowany jest podwyższoną jaskrawością. Po powrocie z podprogramu zmienna **AW** wskazuje adres następnego wiersza. Aby go zanalizować, wystarczy tylko ponownie wywołać podprogram przez **GO SUB 983Ø**. Dla ułatwienia sobie życia można też sporządzić krótki program pomocniczy:

Program 58

```
1 INPUT "Numer wiersza:";NW: GO SUB 9850
2 GO SUB 9830: INPUT "Nast. wiersz";Q$
3 IF Q$="" THEN GO TO 2
4 STOP
```

Gdy w odpowiedzi na pytanie "Nast.wiersz" wciśniemy ENTER, zostanie wyświetlony kolejny wiersz.
W podprogramie **LISTUJWIERSZ** najciekawszy jest fragment rozszyfrowujący wartość stałej. Rozpoznaje się ją po poprzedzającym wartość bajcie 14. Następnych 5 bajtów jest przenoszonych w pole wartości zmiennej **BW**, po czym wyświetlona zostanie po prostu sama **BW**. Skąd jednak wiadomo,

gdzie w pamięci znajduje się pole wartości zmiennej **BW**? Wykorzystano w tym przypadku zmienną systemową **DEST** (dwubajtowa, adres: 23629). Podczas operacji przypisania interpreter znajduje w tablicy zmiennych zmienną, która ma otrzymać nową wartość, i zapisuje adres jej pola wartości pomniejszony o 1 w zmiennej **DEST**. Spójrzmy na wiersz 9845 w programie 57. Zmiennej **BW** zostanie przypisana wartość funkcji **FN D**, obliczona ze zmiennej systemowej **DEST**. **BW** otrzyma w wyniku tej operacji wartość przedstawiającą jej własny adres. Ponieważ zamierzamy wpisać w pole wartości **BW** nową zawartość, trzeba przechować ten adres w zmiennej **ABW**. Podany sposób można wykorzystać z powodzeniem także w innych przypadkach, gdy potrzebny nam jest adres zmiennej. Sytuacja taka może wystąpić np. przy przekazywaniu parametrów do procedur maszynowych lub w przekazywaniu wyników ich działania zmiennym języka BASIC.
Znając adresy linii programu możemy często uniknąć kłopotów z lokalizacją programów maszynowych — zwłaszcza gdy są niezbyt długie i relokowalne. Można je wbudować w program w języku BASIC, umieszczając je w instrukcjach komentarza **REM**. Takie procedury maszynowe można przechować w pamięci zewnętrznej wraz z programem pojedynczą instrukcją **SAVE** i następnie w podobny sposób załadować. Spróbujmy wpisać w instrukcję **REM** np. procedurę do inwersji ekranu (program 53). Potrzebny będzie nam: program ładujący (program 46), blok danych (program 53) oraz podprogram **SZUKAJWIERSZ** (program 56). Najpierw zarezerwujemy w programie miejsce dla programu maszynowego, wprowadzając instrukcję **REM** z dowolnym numerem i taką liczbą dowolnych znaków po **REM**, jaka jest liczba bajtów w procedurze maszynowej. Procedura "inwersja" liczy 16 bajtów:

56Ø REM Ø123456789Ø12345

Teraz zleceniem:

LET NW =56Ø: GO SUB 985Ø: PRINT AW

ustalamy adres instrukcji 56Ø w pamięci. Uwzględnimy nagłówek i dodamy do wyświetlonej wartości 4, po czym otrzymaną wartość wpiszemy w pierwszą linię bloku danych jako adres ładowania; np.:

9661 DATA 23759, 9661

Adres ten jest równocześnie wartością **adrp**. Teraz można już uruchomić program ładujący i wypróbować załadowany program zleceniem np.:

RANDOMIZE USR 23759

Próba listowania takiej zmodyfikowanej linii **REM** prowadzić może często do błędów: zawarte w niej kombinacje znaków mogą być nielegalne w normalnym programie. Nie prowadzi to jednak do jakichkolwiek zaburzeń w pracy programu.

Ładowanie programów maszynowych do linii **REM** powinno odbywać się już po nadaniu programowi ostatecznej postaci, tak aby linie te nie były przemieszczane. Można to osiągnąć także umieszczając je na samym początku programu. Ewentualne przemieszczenia nie mają znaczenia dla programów relokowalnych. Przed każdym użyciem po modyfikacji programu trzeba jednak na nowo ustalić ich adresy, np. podprogramem **SZUKAJ-WIERSZ**. Linie **REM** z relokowalnymi procedurami maszynowymi można także zapamiętać oddzielnie na taśmie i utworzyć z nich bibliotekę programów maszynowych, dołączanych w razie potrzeby w gotowej postaci przez **MERGE**.

Połączenie ze światem zewnętrznym

Ze wszystkich systemowych elementów języka BASIC pozostały nam do omówienia: instrukcja **OUT** i funkcja **IN**. Gdy mikrokomputer chce przekazać do otoczenia jakąś informację, posługuje się tzw. bramą wyjściową (portem). Teoretycznie w ZX Spectrum może być 256 (a nawet 65535) różnych, ośmiobitowych bram. Każda z takich bram wymaga oczywiście odpowiedniej przystawki sprzętowej. Instrukcja **OUT** wyprowadza do bramy o określonym numerze bajt informacji:

OUT 254, Ø

Brama (port) nr 254 jest wewnętrzną bramą (systemową) ZX Spectrum. Tą drogą wyprowadzana jest informacja dla magnetofonu i głośniczka, a także kolor ramki. Po wykonaniu powyższej instrukcji kolor ramki zmieni się na czarny, lecz zaraz po pierwszej ingerencji z klawiatury wróci do poprzedniego stanu. Za kolor ramki odpowiadają trzy najmłodsze bity portu. Głośnik obsługuje bit nr 4. Wypróbujmy program: (**BIN ØØØ1ØØØ** = 16)

1Ø OUT 254, Ø : OUT 254, 16 : GO TO 1Ø

W nieskończonej pętli wartość bitu 4 bramy systemowej zmieniana jest z Ø na 1 i na odwrót. Z głośnika wydobywa się więc ton.
Wykonajmy **NEW** i wprowadźmy z kolei program następujący:

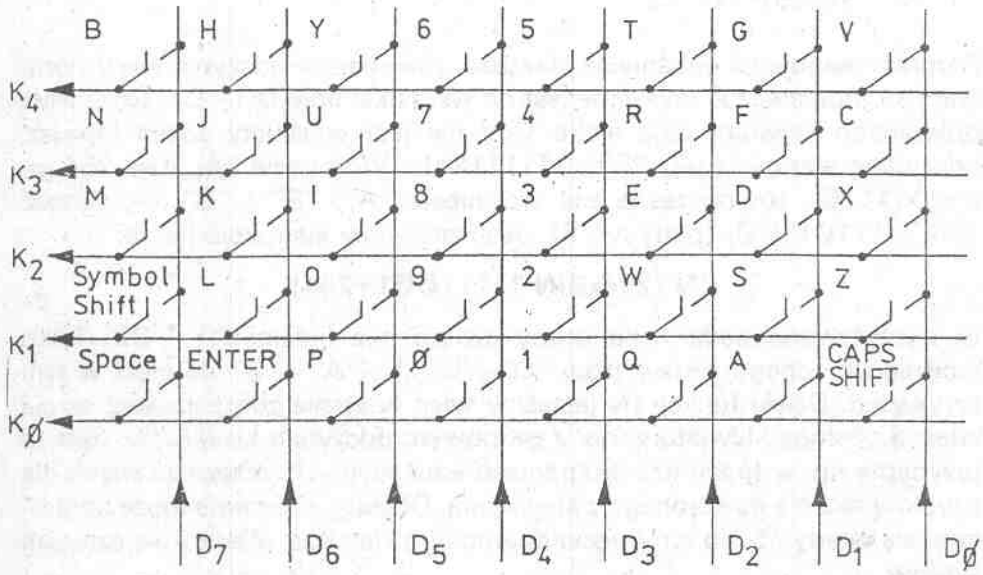
Program 59

```
10 LET X=0
20 LET X=(X=0)*5
30 OUT 254, X
40 GO TO 20
```

Obrazek, który powstanie na ekranie, będzie niezwykły (niespodzianka!). Wciśnięcie dowolnego klawisza zmieni nieco obraz: wynika to z faktu, że obsługa klawiatury wymaga pewnego czasu i tempo pracy komputera przy

wciśniętym klawiszu jest trochę wolniejsze. Spróbujcie zmienić liczbę w linii 2Ø z 5 na 4, 6 lub np. 255. Działanie programu sprowadza się do bezpośredniego wpływu na barwę ramki: barwa ta jest periodycznie zmieniana.

Oprócz bram wyjściowych istnieją także bramy wejściowe. Do odczytania informacji z tych bram służy funkcja **IN**. Także i tu odczytywany jest pojedynczy bajt. Bram wejściowych może być do 256 (w pewnych warunkach do 65535). Spectrum posiada systemową bramę wejściową o numerze (adresie) 254 — podobnie jak brama wyjściowa. 5 najmłodszych bitów tej bramy wykorzystywanych jest do odczytu klawiatury. W jaki sposób można jednak odczytać stan klawiatury złożonej z 40 przycisków za pomocą jedynie 5 bitów? Aby odpowiedzieć na to pytanie, przyjrzyjmy się budowie klawiatury (rys. 7):



Rys. 7 Organizacja klawiatury w ZX Spectrum

Każdy klawisz jest po prostu zwykłym zestykiem, łączącym dwa przewody. Istnieją dwie grupy przewodów: pięć z nich, połączonych każdy z jednym biegunem ośmiu łączników, odczytywanych jest przez port 254. Osiem pozostałych dołączonych jest do drugiego bieguna — po pięć klawiszów każdy. W trakcie odczytu pięciu linii wejściowych można na ośmiu pozostałych liniach stworzyć dowolną kombinację bitów. Jeśli np. ustawimy Ø tylko na jednej z linii, na pozostałych zaś — poziom wysoki (logiczne 1), to w razie wciśnięcia któregoś z klawiszów wybranej grupy łatwo ustalimy, o który chodzi: odpowiadająca mu linia wejściowa będzie w stanie Ø (linie nie podłączone mają potencjał wysoki — logiczne 1). Jeśli powtórzymy tę

operację ośmiokrotnie — za każdym razem umieszczając Ø na innej linii, to będziemy w stanie zidentyfikować każdy wciśnięty klawisz. Jak ustawić na ośmiu liniach wyjściowych zadaną kombinację bitów? Wchodzenie w szczegóły pracy mikroprocesora nie jest naszym zamiarem. Wystarczy wiedzieć, że argumentem funkcji **IN** może być liczba szesnastobitowa i że obydwa jej bajty mają zupełnie inne znaczenie. Młodszy jest numerem portu, starszy zaś zawiera tę kombinację bitów, która zostanie umieszczona na liniach DØ—D7 klawiatury w czasie odczytu. Napiszmy krótki program:

Program 60

```
10 LET x=IN 254
20 LET Y=IN 254: IF X=Y THEN GO TO 20
30 PRINT Y: LET X=Y: GO TO 20
40 GO TO 20
```

Program reaguje na wciśnięcie klawisza, zmieniające odczytywaną z portu wartość. Ponieważ Ø wysyłane jest na wszystkie linie DØ—D7, wciśnięcie dowolnego klawisza daje efekt. Gdy nie jest wciśnięty żaden klawisz, odczytaną wartością jest $255_{10}=11111111_2$. Wciśnięcie "A" daje $254_{10}=11111110_2$, równoczesne zaś wciśnięcie "A", "S" i "G" — wartość $236_{10}=111Ø11ØØ_2$ (patrz rys. 6). Jeśli zmienimy instrukcje **IN** na:

IN (256•BIN 11111ØØ1+254)

to odczytywane będą tylko grupy związane z liniami D1 i D2. Tylko wciśnięcie jednego z klawiszów "Q"—"T" lub "A"—"G" da efekt w tym przypadku. Dzięki funkcji **IN** jesteśmy więc w stanie zorganizować swoją własną obsługę klawiatury, np. z grupowym odczytem klawiszów. Jest to przydatne np. w grach lub programach edukacyjnych, przeznaczonych dla użytkownika nie oswojonego z klawiaturą. Obsługa programu może ograniczać się wtedy np. do wciśnięcia dowolnego klawisza w jednym z czterech rzędów.

W roli użytkownika

Wiemy już sporo o intymnym życiu naszego mikrokomputera. Wiedza ta powinna ułatwić nam racjonalne pisanie własnych programów.

Optymalizacja programów w języku BASIC

Oszczędzanie czasu wykonania i pamięci jest niekiedy wymogiem sytuacji. Optymalizacja programów prowadzi często do znacznego pogorszenia struktury i czytelności. W większości programów najdłużej trwa realizacja fragmentu liczącego nie więcej niż kilka—kilkanaście procent całej objętości. Wystarczy poddać optymalizacji właśnie ten fragment. Największe oszczędności pamięci można na ogół osiągnąć nie użyciem sztuczek, lecz starannym zaprojektowaniem struktury programu i danych. Niekiedy jednak, np. w Spectrum 16 KB, użycie pewnych zabiegów optymalizacyjnych jest koniecznością.

Każda linia programu zawiera oprócz właściwej treści 5 bajtów organizacyjnych. Umieszczając kilka instrukcji w jednej linii można więc sporo zaoszczędzić. Takie postępowanie, prowadzące do zmniejszenia ogólnej liczby linii programu, przyspiesza też wykonanie instrukcji **GO TO**, **GO SUB** i **FOR... NEXT**. Wskazane jest unikanie dużych zbiorów stałych. Każda stała zajmuje od siedmiu do kilkunastu bajtów. Jeśli stała występuje w programie często, warto nadać jej wartość zmiennej prostej i odwoływać się właśnie do tej zmiennej, której nazwa może być przecież pojedynczym znakiem. Szczególnie wskazane jest unikanie stałych dziesiętnych w zbiorach **DATA**. W programach graficznych, muzycznych, edukacyjnych trzeba zwykle pamiętać co najmniej kilkaset liczb, określających współrzędne punktów, dźwięki melodii itp. Korzystnie jest wartości te wpisać w stałe łańcuchowe. Mamy np. do zapamiętania 10 różnych wartości liczbowych z przedziału 0—255. Najprostsze rozwiązanie:

DATA 123, 215, 222, 100, 9, 234, 255, 116, 200, 151

wymaga bez bajtów organizacyjnych i słowa kluczowego **DATA** aż 97 bajtów. Gdy te same dane zapiszemy jako łańcuch:

DATA "12321522210000923425511620151"

zużyjemy tylko 32 bajty (30 cyfr i dwa znaki cudzysłowu). Jedyny problem to odczyt danych. Należy podzielić łańcuch na grupy trzyznakowe i zastosować instrukcję **VAL**

READ L$: LET X=VAL L$(1 TO 3)

Operację taką zazwyczaj będziemy stosowali w pętli. Podobnie przedstawia się sprawa zmiennych. Programy graficzne wykorzystują często duże tablice do przechowywania współrzędnych punktów. Uwzględniając rozmiary ekranu, współrzędne te są jednak często liczbami nie przekraczającymi 255. Można je przechowywać w pojedynczych bajtach, np. w elementach jedno- lub wielowymiarowej tablicy łańcuchowej. Zamiast deklarować tablicę:

DIM X(1000): DIM Y(100, 10)

wystarczy zadeklarować:

DIM X$(1000): DIM Y$(100, 10)

Jedyny kłopot to konieczność zapisu i odczytu wartości poprzez funkcje **CHR$** i **CODE:**

Zamiast: **LET X(215) = 125** zapiszemy **LET X$(215)=CHR$(125)**
Instrukcję: **LET V=X(57)** zastąpimy przez **LET V=CODE X$(57)**

Kosztem niewielkiej komplikacji programu zredukowaliśmy potrzebną dla danych pamięć do 20% początkowej objętości. Jeśli trzeba pamiętać większe liczby całkowite, należy dla jednej wartości zarezerwować po dwa sąsiednie bajty zmiennych tekstowych. Można też użyć zmiennej tekstowej o elementach dwubajtowych:

DIM X$ (1000, 2)

Zamiast: **LET V=X(35)** zapiszemy **LET V=X$(35, 1)+256*X$(35, 2)**

Wygodna może okazać się odpowiednia, zadeklarowana funkcja. Zapis danych w tablicach łańcuchowych często jest wygodniejszy od posługiwania się instrukcją **POKE** i funkcją **PEEK**. Nie grozi kolizja obszarów pamięci, dane można też zapisać na taśmę wraz z programem instrukcją **SAVE**.
Więcej uwagi warto poświęcić optymalizacji czasu wykonywania programu. Zalecane tu środki często stoją w sprzeczności z zasadami optymalizacji objętości programu. Celowe jest przedstawianie wszystkich stałych bezpośrednio w treści programu. Wyszukiwanie zmiennych w tablicy zmiennych wymaga czasu. Należy unikać w miarę możliwości czasochłonnej operacji potęgowania, zastępując ją mnożeniem tam, gdzie jest to wykonalne. W programach graficznych często potrzebne są funkcje trygonometryczne,

głównie **SIN** i **COS**. Także i te funkcje obliczane są długo. Najczęściej jednak nie jest nam potrzebna zbyt wielka dokładność i wystarczają wartości funkcji dla np. pełnych stopni. Można wtedy wstępnie obliczyć wartości funkcji dla kątów w mierze stopniowej od 1 do 36Ø, zapisując je np. w tablicy **S**. Aby później w programie wyznaczyć sinus kąta 55 stopni, wystarczy szybko realizowana instrukcja:

LET sinus=S(55)

Przy użyciu odpowiednich zależności trygonometrycznych można objętość tablicy ograniczyć do np. 9Ø elementów, dla kątów od Ø do 89 stopni, a funkcje dla kątów z pozostałych ćwiartek układu współrzędnych wyznaczać ze znanych wzorów.
Powodem poważnych strat czasu jest w większych programach przeszukiwanie tablicy zmiennych, wyszukiwanie numerów instrukcji przy skokach, poszukiwanie deklaracji funkcji po jej wywołaniu.
Pamiętajmy, że zmienne, które wystąpiły w programie jako pierwsze, zostaną umieszczone na początku tablicy i będą dostępne szybciej od zmiennych umieszczonych na końcu. Dlatego dobrze jest ustalić, które zmienne wykorzystywane są najczęściej (np. zmienne sterujące pętli **FOR... NEXT**) i zmienne te zadeklarować na samym początku programu instrukcją przypisania, np.:

1Ø LET X=Ø : LET Y=Ø

Zmiennych sterujących w ten sposób zadeklarować się nie da: zmienne te zapisywane są w innym formacie i wymagają więcej pamięci. Jeśli nawet umieścimy w programie instrukcję:

15 LET I=Ø

a później w programie użyjemy zmiennej **I** jako zmiennej sterującej, to stary zapis zostanie z tablicy usunięty i zmienna **I** znajdzie się na końcu. Wyjściem jest pusta pętla na początku programu:

15 FOR I=1 TO 1: NEXT I

Teraz zmienna **I** znajdzie się na początku tablicy i żadna siła jej stamtąd nie ruszy. Kłopotliwa jest sprawa zmiennych łańcuchowych prostych: po każdym przypisaniu zmienna przepisywana jest na koniec tablicy, a przesuwanie pozostałych zmiennych na jej miejsce powoduje dodatkowe straty czasu. Lepiej wtedy użyć zmiennych łańcuchowych deklarowanych, sztywnej długości. Ich miejsce w tablicy zmiennych jest stałe, można więc umieścić je w razie potrzeby na początku tablicy. W długim programie często używane funkcje i podprogramy należy umieścić w miarę możliwości w pierwszych jego liniach. Gdy po wywołaniu interpreter zacznie przeszukiwać program od początku w poszukiwaniu deklaracji funkcji lub numeru wiersza podprogramu, to prędko natknie się na szukany obiekt. To samo dotyczy krytycznych

czasowo pętli **FOR... NEXT**: pamiętajmy, że w Spectrum instrukcja **NEXT**, inaczej niż w innych mikrokomputerach, wykonywana jest podobnie jak **GO TO**: odbywa się poszukiwanie numeru wiersza i instrukcji.

Usprawniamy ZX Spectrum

Nasz mikrokomputer jest wygodny w użyciu. Mimo to ma wiele słabości i mankamentów. Niekiedy ich usunięcie jest proste. Możliwe jest też wyposażenie Spectrum we właściwości dostępne w innych mikrokomputerach.

1. **Akustyczna kontrola klawiatury.** Słyszalność sygnału akustycznego przy wciśnięciu klawisza można polepszyć, wpisując do komórki 236Ø9 wartość rzędu 5—2Ø (trzeba poeksperymentować). Komórka ta mieści zmienną systemową **PIP**, określającą liczbę cykli sygnału przy każdym wciśnięciu klawisza.

2. **Zwiększenie prędkości kursora.** Po nabyciu wprawy w korzystaniu z klawiatury prędkość przemieszczania się kursora przy edycji linii może okazać się za wolna. Czas, jaki upłynie między kolejnymi ruchami kursora, wyrażony w pięćdziesiątych częściach sekundy, określa zmienna systemowa **REPPER** (adres: 23562). Początkowo zawartość tej komórki wynosi 5. Można ją zredukować do wartości 2—3.

3. **Zmiana kursora w instrukcji INPUT.** Wiele mikrokomputerów w roli znaku zachęty w instrukcji **INPUT** używa znaku zapytania. ZX Spectrum nie ma specjalnego znaku zachęty. Obecny w oknie systemowym kursor "L" nie zawsze jest dostatecznie widocznym symbolem oczekiwania na wprowadzenie informacji. Poprzez **POKE 23617, 236** można spowodować, że kursor w instrukcji **INPUT** przyjmie postać pytajnika. Niestety, po ewentualnym błędzie przy wprowadzaniu danych kursor powraca do pierwotnej postaci. Dlatego omawianą instrukcję **POKE** warto umieścić oddzielnie przed każdą instrukcją **INPUT**.

4. **Określenie objętości wolnej pamięci.** Najprościej posłużyć się tu zleceniem:

PRINT 65535−USR 7962

Wykorzystywana jest procedura maszynowa zawarta w pamięci ROM.

5. **Usprawnienie zapisu danych na taśmie.** Zapis danych na taśmie nie jest w Spectrum najwygodniejszy (niemożliwość zapisu zmiennych prostych itp.). Jeśli zapisujemy kolejno kilka tablic, to przed rozpoczęciem zapisu

każdej z nich komputer wyświetli komunikat "Start tape, then press any key" (Włącz magnetofon, potem naciśnij dowolny klawisz) i będzie oczekiwał na wciśnięcie klawisza. Zmusza to do ciągłej uwagi przy zapisie. Uniknąć komunikatu i oczekiwania na klawisz można poprzez instrukcję **POKE 23736, 187**, poprzedzającą odpowiednią instrukcję **SAVE**. W praktyce warto pozwolić na komunikat przed pierwszą operacją zapisu. Pozwoli nam to uruchomić magnetofon. Następne mogą już odbywać się automatycznie.

```
4560 SAVE "TAB1" DATA X()
4570 POKE 23736,187: SAVE "TAB2" DATA Y()
4580 POKE 23736,187: SAVE "TAB3" DATA T$()
```

Jeśli trzeba zapisywać większą liczbę oddzielnych danych, to powyższy sposób jest nieekonomiczny. Przed każdą tablicą, nawet niewielką, wysyłany jest na taśmę oddzielny nagłówek. Wydłuża to zapis i odczyt. Zapamiętanie wszystkich danych wraz z programem nie wchodzi w rachubę w przypadku długich programów. Korzystne byłoby zapisanie na taśmie samej tylko tablicy zmiennych. Instrukcja **SAVE** wysyła na taśmę ciągły obszar pamięci od wskazywanego przez zmienną systemową **PROG** początku programu do końca tablicy zmiennych. Jeśli na czas operacji zapisu zastąpimy wartość **PROG** przez **VARS** (adres początku tablicy), to na taśmie zapamiętane zostaną wyłącznie zmienne:

Program 61

```
9980 LET q=PEEK 23635: POKE 23635,23627
9981 LET q1=PEEK 23636: POKE 23636,23628
9982 SAVE "Nazwa"
9983 POKE 23635,q1: POKE 23636,q2
9985 STOP
```

Zmienne ładujemy z taśmy instrukcją **MERGE "Nazwa"**. Zapisanych w wyżej opisany sposób danych nie można niestety zweryfikować w prosty sposób przez **VERIFY**.

6. **Symulacja instrukcji GET**. Niekiedy wprowadzamy informację znak po znaku i chcemy rejestrować pojedyncze wciśnięcia klawisza, nie zaś bieżący stan klawiatury. Wiele mikrokomputerów oferuje tu instrukcję **GET**. Dostępna w Spectrum funkcja **INKEY$** musi być umieszczona w pętli, np.:

```
300 LAT A$=INKEY$: IF A$="" THEN GO TO 300
```

Jest to niewygodne, trudno jest regulować prędkość wprowadzania. Prościej i wygodniej użyć instrukcji **PAUSE 0**:

300 PAUSE 0 : LET A$=INKEY$

Wykonywanie instrukcji **PAUSE 0** kończy się w chwili naciśnięcia dowolnego klawisza. Odpowiadający mu znak zostanie natychmiast odczytany przez funkcję **INKEY$**.

7. **INPUT w dowolnym miejscu ekranu.** Większość mikrokomputerów pozwala na wprowadzanie danych instrukcją **INPUT** w dowolnym miejscu ekranu. Wprowadzone z klawiatury znaki pozostają na ekranie. W Spectrum miejscem wprowadzania danych jest okno systemowe, a po wciśnięciu ENTER zawartość okna znika. Jest to niekiedy kłopotliwe przy wprowadzaniu np. szeregu kolejnych danych. Poniższy podprogram wprowadza łańcuch znaków i może pracować w dowolnym miejscu ekranu, wskazanym przez ostatnio wykonaną instrukcję **PRINT**. Wprowadzone znaki pozostają na ekranie. Respektowany jest klawisz DEL. Wprowadzanie łańcucha kończy się — jak w **INPUT** — klawiszem ENTER. Po powrocie z podprogramu zmienna **V$** zawiera wprowadzony łańcuch. Jeśli miała być wprowadzona wartość liczbowa, wystarczy użyć funkcji **VAL**:

LET V=VAL V$

Rozpoznawany w linii 9827 kod 12 sygnalizuje wciśnięcie klawisza DEL. Wtedy z łańcucha **V$** usuwany jest ostatni znak. Aby znak ten usunąć z ekranu, wskaźnik cofany jest w lewo znakiem **BS**, w miejsce kasowanego symbolu wyświetlana jest spacja, po czym wskaźnik ponownie cofany jest znakiem **BS**:

Program 62

```
9820 REM NINPUT(V$)
9821 LET V$=""
9822 PAUSE 0: LET q$=INKEY$
9823 IF q$>CHR$ 164 THEN  GO TO 9822
9824 IF q$<" " THEN  GO TO 9827
9825 LET V$=V$+q$: PRINT q$;
9826 GO TO 9822
9827 IF q$=CHR$ 13 THEN  PRINT : RETURN
9828 IF q$=CHR$ 12 THEN  IF LEN V$>0 THE
N  LET V$=V$( TO LEN V$-1): PRINT CHR$ 8
; " ";CHR$ 8;
9829 GO TO 9822
```

8. **PRINT USING**. Format wyprowadzania liczb instrukcją **PRINT** jest niezbyt elastyczny. Zwłaszcza adiustacja formatu do lewego marginesu jest niezgodna z przyzwyczajeniami. Przy wyprowadzaniu kolumn liczb np. w tabelach ważne jest, aby odpowiadające sobie pozycje były umieszczone pod sobą, a wszystkie liczby zostały przedstawione z taką samą liczbą miejsc

po przecinku. Wiele mikrokomputerów dysponuje zmodyfikowaną **PRINT USING**, w której można zdefiniować postać wyświetlanych lub drukowanych liczb. Brak tej możliwości w ZX Spectrum rekompensuje funkcja **U$**, zdefiniowana następująco:

```
9914 DEF FN U$(X,D,U)="
( TO D-(X<0)-LEN (STR$ INT ABS X)-U-1)+"
-"( TO X<0)+STR$ (INT ABS X)+(STR$ (ABS
X-INT ABS X+1+.5*10↑-U))(2 TO 2+U)
```

Łańcuch spacji, umieszczony na początku definicji, liczy dokładnie 16 znaków. Funkcja ma trzy argumenty. Pierwszym jest argument liczby, której wartość chcemy wyprowadzić. Drugi określa całkowitą długość pola, zajmowanego przez liczbę (do 16). Trzeci argument definiuje liczbę cyfr po kropce dziesiętnej (do 6). Cyfry dosunięte są do prawego krańca pola zadanej szerokości. Przy wyprowadzaniu kolumn liczb odpada więc stosowanie modyfikatora **TAB**. W poniższym przykładzie wyświetlana jest tablica kwadratów i sześcianów, przy czym poszczególne kolumny mają różne formaty:

Program 63

```
10 FOR X=1 TO 3 STEP 0.1
20 PRINT FN U$(X,6,1); FN U$(X↑2,8,2):FN U$(X↑3,12,4)
30 NEXT X
```

9. **ON... GO TO** i **ON... GO SUB**. Wiele dialektów języka BASIC dysponuje tzw. skokami liczonymi: zwykłymi i do podprogramów. Instrukcja skoku zawiera wiele możliwych adresów skoku, ich wybór zaś odbywa się w zależności od wartości wyrażenia sterującego:

```
ON I GO TO 100, 230, 500
```

Gdy **I=1**, skok nastąpi do linii 100, I=2 — do 230, I=3 — do 500. W razie potrzeby adresów skoku można wyszczególnić więcej. Jeśli wartość wyrażenia sterującego jest mniejsza od 1 lub większa od ilości dostępnych numerów wiersza, w większości dialektów skok zostanie zignorowany i program przejdzie do następnej instrukcji. W Spectrum nie ma instrukcji **ON... GO TO** i **ON... GO SUB**. Można je jednak łatwo zasymulować używając wyrażeń logicznych:

```
IF I>0 AND I<4 THEN GO TO (I=1)*100+(I=2)*230+(I=3)*500
```

W danej chwili tylko jedno z wyrażeń logicznych może być prawdziwe i tylko jeden z członów wyrażenia decyduje o adresie skoku.

10. **ON ERROR GO TO**. Wystąpienie jakiegokolwiek błędu w trakcie wykonania programu powoduje w ZX Spectrum jego nieuchronne przerwanie połączone z wyświetleniem komunikatu. Zdarzają się sytuacje, gdy zaistnienie błędu nie powinno przerywać programu, lecz uruchamiać odpowiednią reakcję programu. Tak jest np. w przypadku błędów podczas wprowadzania danych. Liczne mikrokomputery dysponują w języku BASIC instrukcją **ON ERROR GO TO...**, która powoduje skok do podanego adresu w razie wystąpienia jakiegokolwiek błędu. Brak tej instrukcji w ZX Spectrum można zrekompensować używając specjalnej procedury maszynowej:

Program 64

```
9690 REM ON ERROR GOTO
9691 DATA 23296,9691
9692 DATA "110A5B2A3D5C73237241"
9693 DATA "C9Ø1401F113F5BFD7E4F"
9694 DATA "ØØ3C121321425C712334"
9695 DATA "702336Ø123Ø1Ø3ØØEDDE"
9696 DATA "B0E5FD3600FFFE0C20F1"
9697 DATA "04FD3601CC21761BE59B"
9698 DATA "2A3D5C3603233613C931"
9699 DATA ""
```

Procedura nie jest relokowalna, jednak dzięki zlokalizowaniu jej w buforze drukarki (komórki 23296—23362) nie powinna kolidować z innymi programami. Uruchomienie odbywa się przez **RANDOMIZE USR 23296**. Od tego momentu po jakimkolwiek błędzie nastąpi skok do linii numer 8ØØØ. W linii tej powinien zaczynać się program obsługi błędu w języku BASIC. Numer błędu — zgodny z obowiązującą w Spectrum numeracją — można odczytać w komórce 23359. Numer linii, w której wystąpił błąd, zawierają bajty 2336Ø i 23361. Numer instrukcji w wierszu przechowany jest w bajcie 23362. Tak więc procedura obsługi błędu dysponuje pełną informacją o miejscu wystąpienia i charakterze błędu. Po każdym błędzie trzeba procedurę maszynową uaktywnić ponownie przez **RANDOMIZE USR 23296**. Wyłączenie procedury możliwe jest przez **RANDOMIZE USR 2335Ø**. Jeśli chcemy, aby po błędzie następował skok do innej linii niż 8ØØØ, wystarczy zmienić numer linii zawarty w komórkach 233Ø7 i 233Ø8. Szczegóły korzystania z procedury ukazuje poniższy program przykładowy:

Program 65

```
10 RANDOMIZE USR 23296
20 LET A=X
30 LET W=30/0
40 LET Y=200: PLOT 30,Y
```

```
  50 NEXT Y
  60 LET X=1: LET Y=1: READ Q
  70 STOP
  80
8000 LET NRB=PEEK 23359
8010 PRINT "BLAD NR ";NRB;TAB 13;
8020 LET LIN=PEEK 23360+256*PEEK 23361
8030 PRINT " W LINII ";LIN;"/";PEEK 23362
8040 PAUSE 100: IF NRB=9 THEN STOP
8050 RANDOMIZE USR 23296: GO TO LIN+10
```

W programie przykładowym zaprogramowano wystąpienie kilku błędów. Ponieważ jednak procedura **ON ERROR GO TO** jest uaktywniona, po każdym błędzie nastąpi skok do linii 8ØØØ. Tam wyświetlony zostanie numer błędu, a zmienna **LIN** przyjmie wartość równą numerowi wiersza z błędem. Po wyświetleniu numeru wiersza i instrukcji, w której błąd wystąpił, i stwierdzeniu, że kod błędu jest różny od 9 (STOP), następuje ponowne uaktywnienie **ON ERROR GO TO** i skok do linii następnej po linii, w której ostatnio wystąpił błąd.

11. **"Okrąglejsze" kółko**. Algorytm wykreślania okręgu na ekranie ZX Spectrum pozostawia nieco do życzenia. Kształt okręgu nie jest najdoskonalszy — szczególnie razi brak symetrii. Szybkość kreślenia także nie jest rewelacyjna. Mankamenty te wynikają z faktu, że Spectrum składa okrąg z odcinków prostej, co wymaga żmudnych przeliczeń i powoduje błędy zaokrągleń. Wyjściem jest zastosowanie innego algorytmu kreślenia okręgu. Tam, gdzie szybkość nie jest najistotniejsza, wystarczy poniższy program w języku BASIC:

Program 66

```
9810 REM OKRAG(x0,y0,r) wg Romana Lowkisa
9811 LET x=0: LET da=INT (r/2)
9812 IF X>r THEN RETURN
9813 IF da<0 THEN LET da=da+r: LET r=r-1
9814 LET da=da-x-1
9815 PLOT x0+x,y0+r: PLOT x0-x,y0+r
9816 PLOT x0+x,y0-r: PLOT x0-x,y0-r
9817 PLOT x0+r,y0+x: PLOT x0-r,y0+x
9818 PLOT x0+r,y0-x: PLOT x0-r,y0-x
9819 LET x=x+1: GO TO 9812
```

Przed wywołaniem podprogramu nadajemy zmiennym **XØ** i **YØ** wartości odpowiadające współrzędnym środka okręgu, zmiennej zaś **r** — jego promieniowi. Podprogram OKRAG kreśli równocześnie wszystkie ćwiartki

okręgu. Dzięki temu narysowany okrąg cechuje się idealną symetrią. Zwróćmy uwagę, że wyznaczanie kolejnych punktów okręgu odbywa się bez użycia funkcji trygonometrycznych, a jedynie przy zastosowaniu najprostszych operacji arytmetycznych! Dzięki temu okrąg rysuje się dość szybko nawet w języku BASIC. Dla tych Czytelników, którzy lubią zaskakujące i szybkie efekty, przydatniejsza będzie jednak maszynowa realizacja przedstawionego powyżej algorytmu:

Program 67

```
9700 REM ULEPSZONY OKRAG
9701 DATA 23363,9701
9702 DATA "0000000ED5B445BED4B1F"
9703 DATA "435BAF5F7A1F677BBAE1"
9704 DATA "D07CA7F25D5B82153D71"
9705 DATA "93F5D5C5CD6B5BC1D147"
9706 DATA "E11C18E8C5CD735BC11E"
9707 DATA "7B5A57C579834FCD7E87"
9708 DATA "5BC179934FC57882477D"
9709 DATA "CD895BC1789247CDAA3A"
9710 DATA "223C473E010F10FDB6B6"
9711 DATA "77C940","
```

Także i ten program jest nieprzesuwny. Został on też zlokalizowany w buforze drukarki, w sposób nie kolidujący z **ON ERROR GO TO**. Zajmuje komórki 23363—23445. Wywołanie od adresu 23366. Przed wywołaniem do komórki 23363 wpisujemy współrzędną X środka okręgu, a do komórki 23364 — współrzędną Y. W komórce 23365 umieszczamy promień okręgu. Procedura nie tylko rysuje doskonalsze okręgi od tych tworzonych wbudowaną instrukcją **CIRCLE**, ale na dodatek działa nieporównanie szybciej. Łatwo się o tym przekonać chociażby poniższym programem przykładowym:

```
10 POKE 23363,128: POKE 23364,87
20 FOR R=1 TO 87 STEP 3
30 POKE 23365,R: RANDOMIZE USR 23366
40 NEXT R
```

Oczywiście przed uruchomieniem programu przykładowego trzeba umieścić w pamięci kod maszynowy procedury!
Powyższy przykład powinien wyleczyć nas ostatecznie z przekonania o absolutnej doskonałości naszego komputera. Posiada on wiele zalet i jest rezultatem wytężonej pracy specjalistów. Mimo to możemy sami znacznie go ulepszyć.

12. **Kreślenie odcinków przy użyciu współrzędnych absolutnych.** Instrukcja **DRAW** kreśli odcinki wykorzystując współrzędne względne. Często jednak interesuje nas poprowadzenie linii od ostatnio wypełnionego punktu do konkretnego punktu na ekranie. Najwygodniej posłużyć się wtedy zmiennymi systemowymi **COORDX** (adres: 23677) oraz **COORDY** (adres: 23678), zawierającymi współrzędne ostatnio wypełnionego przez **PLOT** lub **DRAW** punktu. Niech X i Y przedstawiają współrzędne punktu docelowego:

DRAW X-PEEK 23677, Y-PEEK 23678

Wykorzystanie zmiennych systemowych zwalnia nas od wprowadzania dodatkowych zmiennych i ma tę zaletę, że niweluje błędy zaokrągleń, które niekiedy potrafią skumulować się do widocznych na ekranie luk, np. w konturze, który teoretycznie powinien się zamknąć.

13. **Kreślenie linii przerywanych.** Instrukcja **DRAW** wykreśla niestety tylko linie ciągłe. Przy ilustracjach, wykresach itp. przydaje się jednak często linia punktowa. Najprostszym wyjściem będzie napisanie własnej procedury, kreślącej odcinki prostej na podobieństwo instrukcji **DRAW**, lecz pozwalającej na swobodny wybór rodzaju kreślonej linii:

Program 68

```
9790 REM NDRAW(DX,DY,DL)
9791 LET X0=PEEK 23677: LET Y0=PEEK 23678
9792 LET Q1=DL: IF DX=0 THEN GO TO 9797
9793 LET Q=SGN DX
9794 FOR U=X0 TO X0+DX STEP Q
9795 LET V=INT (DY/DX*(U-X0)+.5)+Y0
9796 GO SUB 9802: NEXT U: RETURN
9797 IF DY=0 THEN RETURN
9798 LET Q=SGN DY
9799 FOR V=Y0 TO Y0+DY STEP Q
9800 LET U=INT ((V-Y0)*DX/DY+.5)+X0
9801 GO SUB 9802: NEXT V: RETURN
9802 IF Q1>=0 THEN PLOT U,V
9803 LET Q1=Q1+1: IF Q1>=DL THEN LET Q1=-DL
9804 RETURN
```

Podprogram **NDRAW** ma trzy parametry. **DX** i **DY** to względne współrzędne końca w stosunku do początku odcinka — podobnie jak w instrukcji **DRAW**. Zmienna **DL** decyduje o charakterze kreślonej linii. Jeśli **DL**=Ø, kreślona jest linia ciągła. Przy **DL**>Ø kreślona jest linia przerywana złożona z odcinków długości **DL**. Początek odcinka wypada, jak w **DRAW**, w ostatnio wyświetlonym punkcie.

Edytor ekranowy

Komputer jest wspaniałym narzędziem do redagowania tekstów pod warunkiem, że jest wyposażony w odpowiednie oprogramowanie. Redagowanie poszczególnych linii tekstu, a następnie składanie ich w całość jest najprostsze, lecz niewygodne. Znacznie lepiej jest operować bezpośrednio na dużym wycinku tekstu, obejmującym np. pole całego ekranu. Naprowadzamy kursor na miejsce, w którym chcemy dokonać korekty, i bezpośrednio wpisujemy znaki w wybrane miejsce ekranu, usuwamy je lub zastępujemy innymi. Po zakończeniu edycji (redagowania) po prostu odczytujemy zawartość ekranu i przekazujemy ją do dalszego wykorzystania np. zmiennej tekstowej.
Wiele komputerów, np. C-64, dysponuje wbudowanym tzw. edytorem ekranowym, programem pozwalającym formować teksty na ekranie w sposób bezpośredni, interakcyjny. Spectrum możliwości takiej nie ma, można ją jednak stworzyć pisząc odpowiedni program np. w języku BASIC. W wersji minimalnej program taki powinien pozwalać na swobodne przemieszczanie kursora po ekranie i wpisywanie znaków w wybrane miejsce. Pożądana byłaby też reakcja na klawisz ENTER, która powinna polegać na przejściu do początku nowego wiersza. Poniższy program jest właśnie przykładem takiego prostego edytora ekranowego.

Program 69a

```
1000 REM PROSTY EDYTOR EKRANOWY
1010 LET X=0: LET Y=0: LET X0=0: LET Y0=1
1020 LET T$="": BORDER 6: GO TO 1130
1030 PAUSE 0: LET K$=INKEY$: LET K=CODE K$
1040 IF K<31 OR K>127 THEN GO TO 1060
1050 PRINT AT Y,X;K$: LET K=9
1060 IF K=8 AND X>0 THEN LET X=X-1
1070 IF K=9 AND X<31 THEN LET X=X+1
1080 IF K=10 AND Y<21 THEN LET Y=Y+1
1090 IF K=11 AND Y>0 THEN LET Y=Y-1
1100 IF K=13 AND Y<21 THEN LET X=0: LET Y=Y+1
1110 IF K=15 THEN GO TO 1200
1120 IF X0=X AND Y0=Y THEN GO TO 1140
1130 PRINT AT Y0,X0; OVER 1; PAPER 7;" ";
1140 PRINT AT Y,X; OVER 1; PAPER 5;" ";
1150 LET X0=X: LET Y0=Y: GO TO 1030
1200 FOR Y=21 TO 0 STEP -1
1210   FOR X=31 TO 0 STEP -1
1220     LET T$=SCREEN$ (Y,X)+T$
1230   NEXT X
1240 NEXT Y
```

Zmienne **X** i **Y** przedstawiają aktualne, zaś **XØ** i **YØ** — poprzednie położenie kursora. Program jest właściwie pętlą oczekującą na znaki z klawiatury i reagującą na nie. Na ekranie widnieje kursor. Tworzy go pole wypełnione ciemniejszym tłem (**PAPER 5**). Kursor przemieszcza się klawiszami "**5**"—"**8**", wciskanymi równocześnie z SHIFT, lub specjalnymi klawiszami kierunkowymi — w Spectrum+. Kody tych klawiszów (8—11) rozpoznawane są przez instrukcje warunkowe 1Ø6Ø—1Ø9Ø i w przypadku, gdy kursor nie zajmuje już skrajnych położeń, powodują zmianę o 1 odpowiedniej współrzędnej. Linia 11ØØ rozpoznaje kod klawisza ENTER. Jeśli tylko kursor nie znajduje się u samego dołu okna użytkownika, następuje przejście do następnego wiersza wskutek wyzerowania współrzędnej kursora X i zwiększenie o 1 współrzędnej Y. Ruch kursora realizują instrukcje 112Ø—115Ø. Jeśli położenie kursora nie zmieniło się, następuje powrót do początku pętli. W przeciwnym razie odtwarzany jest pierwotny stan pola, w którym kursor znajduje się aktualnie (113Ø), a następnie zmiana tła z białego na błękitne (PAPER 5) w polu nowej lokalizacji (1140). Na koniec współrzędne kursora zapamiętywane są pod postacią zmiennych **XØ** i **YØ**. Zastosowano tu inny sposób modyfikacji atrybutów wybranego pola bez zmiany jego zawartości. Instrukcja **PRINT OVER 1**; " "; nie wpływa w żaden sposób na treść pola. Obraz spacji nie zawiera przecież ani jednego wypełnionego punktu, który mógłby zmienić stan któregoś z punktów już obecnych na ekranie. Modyfikacja atrybutów jednak nastąpi.

Instrukcja 111Ø rozpoznaje kod klawisza GRAPHICS (SHIFT i "9"), kończącego edycję i rozpoczynającego przenoszenie treści ekranu do pustej początkowo zmiennej **T$**. Do zmiennej tej "doklejane" są kolejno znaki odczytywane z poszczególnych pól okna użytkownika funkcją **SCREEN$**. Odczytujemy od dołu ekranu — równie dobrze można było to zrobić od początku i od lewej strony. Trzeba by wtedy zmienić tylko linię 122Ø. Po zakończeniu pracy programu zmienna **T$** przechowuje treść ekranu. Niestety, zmienna ta nie zawiera żadnych informacji o wierszowej strukturze ekranu. Zapamiętane zostały także niepotrzebnie wszystkie spacje położone na prawo od ostatniego znaku w każdym wierszu. Można temu zaradzić, modyfikując sposób odczytu ekranu (linie 121Ø—123Ø):

Program 69b

```
1210 LET Q=0: LET T$=CHR$ 13+T$
1211 FOR X=31 TO 0 STEP -1
1212   LET K$=SCREEN$ (Y,X)
1213   IF K$=" " AND Q=0 THEN GO TO 1230
1220   LET Q=1: LET T$=K$+T$
1230 NEXT X
```

Istotną rolę spełnia zmienna pomocnicza **Q**, zerowana przed analizą każdego wiersza. Na początku dopisujemy do zmiennej **T$** znak ENTER (kod 13),

oznaczający koniec wiersza, po czym badamy poszczególne znaki zaczynając od końca. Zmienna **Q=Ø** oznacza, że jeszcze nie napotkaliśmy żadnego znaku różnego od spacji. Jeśli przy **Q=Ø** odkryjemy spację, to możemy ją spokojnie zignorować. Wykrycie znaku różnego od spacji powoduje jego dopisanie do zmiennej **T$** i zmianę wartości **Q** z Ø na 1. Od tej chwili każda spacja wpływa już istotnie na obraz wiersza, musi więc zostać przeniesiona do **T$**. Zauważmy, że opisane postępowanie możliwe jest tylko podczas analizy wiersza od końca!

Po zakończeniu pracy programu zmienna **T$** nie zawiera zbędnych spacji na końcach linii. Zamiast nich na końcu każdego wiersza mamy znak ENTER. Taki tekst możemy skierować np. na drukarkę: zostanie przedstawiony, w takiej samej postaci, jak na ekranie monitora.

Nasz miniedytor ekranowy trudno uznać za nadmiernie komfortowy. Przy bardziej złożonych korektach trzeba przepisywać duże fragmenty ekranu. Brakuje m.in. funkcji kasowania pojedynczych znaków tekstu z równoczesnym przesuwaniem na ich miejsce fragmentów tekstu położonych z prawej, przydałaby się też możliwość likwidacji całych wierszy lub rozsuwania tekstu dla wprowadzenia wierszy dodatkowych. W ulepszonej wersji edytora wprowadzimy dodatkową tablicę tekstową **T$ (32*22)**, która będzie spełniać funkcję kopii ekranu. Każdemu polu ekranu odpowiada jedna komórka tablicy **T$**. Każda operacja na ekranie, np. wpisanie znaku, jest uzupełniana odpowiednią modyfikacją zawartości tablicy, by w każdej chwili istniała odpowiedniość pomiędzy **T$** i ekranem.

Sens istnienia **T$** ujawnia się szczególnie po naciśnięciu DEL (kod 12). Najpierw modyfikujemy fragment **T$**, odpowiadający tekstowi od kasowanego znaku do końca wiersza. Fragment jest przesuwany w lewo o 1 pozycję, na końcu zaś dopisywana jest spacja. Następnie skorygowany wiersz wyświetlany jest w całości od nowa.

Podobna operacja nastąpi po EDIT (kod 7). Tutaj jednak tekst od kursora do końca wiersza przesuwany jest w prawo, a w polu kursora pojawia się puste miejsce, w które można wstawić brakujący znak.

Program 70

```
1000 REM EDYTOR EKRANOWY
1010 DIM T$(32*22)
1020 LET X=0: LET Y=0: LET X0=0: LET Y0=1
1030 BORDER 6: POKE 23562,2: GO TO 2000
1040 PAUSE 0: LET K$=INKEY$: LET K=CODE K$
1050 IF K<32 OR K>127 THEN GO TO 1080
1060 PRINT AT Y,X;K$: LET T$(1+X+32*Y)=K$
1070 IF X<31 THEN LET X=X+1: GO TO 2000
1080 IF K=8 AND X>0 THEN LET X=X-1
1090 IF K=9 AND X<31 THEN LET X=X+1
1100 IF K=10 AND Y<21 THEN LET Y=Y+1
```

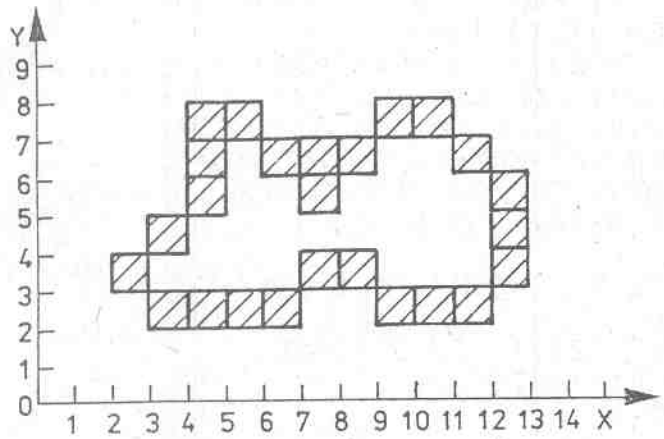
```
1110 IF K=11 AND Y>0 THEN LET Y=Y-1
1120 IF K=13 AND Y<21 THEN LET X=0: LET Y=Y+1
1130 IF K=12 THEN GO TO 2100
1140 IF K=4 THEN GO TO 2200
1150 IF K=5 THEN GO TO 2300
1160 IF K=7 THEN GO TO 2400
1170 IF K=15 THEN POKE 23562,5: RETURN
2000 IF X=X0 AND Y=Y0 THEN GO TO 2020
2010 PRINT AT Y0,X0; OVER 1; PAPER 7;" ";
2020 PRINT AT Y,X; OVER 1; PAPER 5;" ";
2030 LET X0=X: LET Y0=Y: GO TO 1040
2100 IF X=0 THEN GO TO 1040
2110 LET P$=T$(X+32*Y+1 TO 32*Y+32)
2120 LET T$(X+32*Y TO 32*Y+32)=P$+" "
2130 PRINT AT Y,0;T$(32*Y+1 TO 32*Y+32);
2140 IF X>0 THEN LET X=X-1: GO TO 2020
2200 LET T$(1+32*Y TO 704)=T$(33+32*Y TO 704)
2210 PRINT AT 0,0;T$;: GO TO 2020
2300 LET P$="                                "
2310 LET T$(1+32*Y TO )=P$+T$(1+32*Y TO )
2320 GO TO 2210
2400 LET P$=T$(X+32*Y+1 TO 32*Y+31)
2410 LET T$(X+32*Y+1 TO 32*Y+32)=" "+P$
2420 GO TO 2210
```

Jeszcze prostsze jest postępowanie w przypadku kasowania linii (klawisz TRUE VIDEO, kod 4). Cała operacja sprowadza się do przepisania fragmentu **T$** od wiersza położonego poniżej kursora do wiersza, w którym aktualnie znajduje się kursor. Ostatnie 32 znaki zostaną automatycznie zastąpione spacjami, jak zwykle przy operacjach przypisania na łańcuchach sztywnej długości. Następnie zmienna **T$** jest w całości wyświetlana na ekranie. Po wciśnięciu INVERSE VIDEO (kod 5) tekst, począwszy od linii z kursorem, jest przesuwany o 1 linię w dół, a w miejscu rozsunięcia pojawia się pusta linia, w którą można dopisać brakujący tekst. Uwaga! Złożony ze spacji łańcuch w linii 23ØØ musi liczyć dokładnie 32 elementy.
Klawisz GRAPHICS (kod 15) kończy edycję. Zmienna **T$** zawiera zbędne spacje i brakuje w niej znaków ENTER. Jej ostateczne sformowanie nie powinno jednak nastręczać kłopotów: można tutaj postąpić podobnie, jak w przypadku edytora uproszczonego (program 69b).

Zamalowywanie wnętrza konturu

Często występującym zadaniem grafiki komputerowej jest zamalowywanie figury ograniczonej zamkniętym konturem. Przyjmijmy, że kontur może mieć dowolny, nawet zawiły kształt, przedstawiając np. zarys kontynentu. Załóżmy

jeszcze, że do zamalowania wnętrza dowolnego konturu powinno wystarczyć wskazanie jednego, dowolnego punktu należącego do wnętrza ograniczonej konturem figury.
W odróżnieniu od geometrii matematycznej geometria ekranowa operuje punktami o określonych wymiarach i kształcie. Wynika to ze skończonej rozdzielczości ekranu. Za kontur zamknięty uznamy więc taki zamknięty łańcuch punktów, w którym punkty kolejne sąsiadują w dowolny sposób: bokami lub rogami (rys. 8).



Rys. 8 Przykład konturu zamkniętego na ekranie ZX Spectrum

Zamalowywanie wnętrza konturu polega na zapaleniu wszystkich punktów wewnątrz tego konturu. Wzrok człowieka ogarnia z łatwością całość figury. Komputer niestety "widzi" ekran tylko funkcją **PEEK**, może więc w danej chwili odpowiedzieć na pytanie o stan jednego tylko punktu (zapalony lub wygaszony). Aby ustalić wzajemne powiązania kolejno wypełnianych punktów, trzeba zaprojektować odpowiednią organizację (strukturę) danych.
Przy poszukiwaniach najprostszego algorytmu zamalowywania figury pomocne może być przeprowadzenie następującego eksperymentu (najbezpieczniej tylko myślowego!). Wytnijmy z papieru figurę o dowolnym kształcie, a następnie w dowolnym punkcie przytknijmy do niej płonące drewienko. Ogień wkrótce rozprzestrzeni się na całą kartkę, niezależnie od jej kształtu i wyboru punktu podpalenia, chociaż płomień także nie może "widzieć" kartki.
Podzielmy kartkę na umowne, elementarne obszary odpowiadające punktom ekranu. Każdy zapalony punkt kartki zachowuje się identycznie — podpala punkty z nim sąsiadujące. Podstawowa różnica między zamalowywaniem ekranu a spalaniem kartki polega na tym, że komputer może w danej chwili zajmować się jedynie pojedynczym punktem, ogień zaś rozprzestrzenia się po kartce równocześnie w różnych kierunkach. Jak się przekonamy, różnica ta

nie jest zbyt istotna — adaptacja algorytmu "spalania" do zamalowywania ekranu jest całkiem możliwa. W literaturze metoda ta zwana jest często metodą "pożaru prerii".

Załóżmy, że każdy zapalony punkt próbuje zapalić czterech jego najbliższych sąsiadów: z góry, z dołu, z lewej i z prawej. Ponieważ każdy punkt opisany jest parą współrzędnych ekranowych, wskazanie jego sąsiadów jest proste. Trzeba tylko do aktualnych współrzędnych dodać odpowiednio +1 lub −1. Po zapaleniu każdego z sąsiadów będziemy go w przyszłości traktować identycznie, jak punkt wyjściowy itd. Ponieważ jednak w danej chwili możemy zajmować się tylko jednym punktem, współrzędne pozostałych musimy zapamiętać aż do momentu przyszłego wykorzystania. Liczba punktów we wnętrzu figury może iść w tysiące i dziesiątki tysięcy, stając się porównywalną z pojemnością dostępnej pamięci RAM mikrokomputera. Projektując sposób przechowywania współrzędnych nie obsłużonych jeszcze punktów musimy troszczyć się więc o racjonalną gospodarkę pamięcią. Każdy niepotrzebny już punkt powinien być z niej czym prędzej usuwany, robiąc miejsce dla następnych. Kolejnym problemem jest znalezienie kryterium zakończenia pracy i rozpoznania sytuacji, że całe wnętrze figury zostało już zamalowane.

Do przechowywania współrzędnych wykorzystamy strukturę danych zwaną **stosem** programowym (ang. LIFO: least in — first out). Struktura ta zachowuje się podobnie jak zbiór książek ułożonych jedna na drugiej. Do ostatnio położonej książki mamy dostęp bezpośredni, leży wszak na wierzchołku — **szczycie** stosu. Aby dotrzeć do tomu położonego jako pierwszy, a więc leżącego na spodzie (dnie) stosu, trzeba najpierw zdjąć wszystkie woluminy leżące na nim.

Podstawową czynnością programu będzie analiza otoczenia uprzednio wypełnionego (zapalonego) punktu, ściślej — wszystkich czterech jego sąsiadów: z góry, z dołu, z lewej i z prawej. Jeżeli któryś z tych punktów jest już zapalony, to nie interesujemy się nim więcej.

W przeciwnym wypadku punkt ten wypełniamy, a jego współrzędne "układamy na stosie". Gdy zakończymy analizę sąsiadów jednego punktu, przechodzimy do badania otoczenia kolejnego z uprzednio zapalonych punktów. Jego wybór jest zupełnie dowolny. Najprościej będzie więc wybrać punkt znajdujący się na szczycie naszego stosu współrzędnych. Na stosie magazynujemy przecież wyłącznie współrzędne punktów już zapalonych, lecz o nie przebadanym jeszcze otoczeniu. Postępowanie takie jest racjonalne, gdyż po wykorzystaniu współrzędne zdjętego ze stosu punktu możemy odrzucić — nie będą już potrzebne. Układanie współrzędnych na stosie będzie odbywać się na przemian z ich pobieraniem, co ograniczy nadmierny wzrost wysokości stosu. Gdy stos będzie pusty, tzn. nie pozostanie już ani jeden punkt o nie przebadanym otoczeniu, będzie to oznaczało, że nie może być ani jednego punktu wnętrza o nie zapalonych jeszcze sąsiadach. Jednym słowem, całe wnętrze figury zostało już zamalowane. Dla

zapoczątkowania pracy algorytmu wystarczy położyć na pustym początkowo stosie współrzędne punktu podpalenia figury.
W języku BASIC nie ma ani stosu, ani tym bardziej instrukcji realizujących operacje "POŁÓŻ na wierzchołku stosu" i "ZDEJMIJ z wierzchołka stosu". Strukturę danych o właściwościach identycznych ze stosem uzyskamy jednak używając tablicy **S** oraz zmiennej prostej, stanowiącej **wskaźnik stosu WS**. Zmienna ta wskazuje zawsze na tę pozycję tablicy, do której ostatnio wpisaliśmy nową zawartość. Każde odwołanie się do tablicy **S** będzie odnosiło się wyłącznie do pozycji wskazywanej przez **WS**. Przy układaniu na stos **WS** zwiększa się o 1 przed zapisem wartości do tablicy, podczas zaś zdejmowania współrzędnych ze stosu zmienna **WS** zmniejsza się o 1 po odczycie wartości. Poniższy program ilustruje zasadę zamalowywania konturu:

Program 71

```
 10 PLOT 118,90: DRAW 20,0,5.35
 20 DRAW 0,-30: DRAW -20,0: DRAW 0,30
 30 LET X=128: LET Y=50: GO SUB 100
 40 STOP
 50
100 DIM S(300,2): LET WS=0
110 GO SUB 170: IF WS=0 THEN RETURN
120 LET X=S(WS,1): LET Y=S(WS,2): LET WS=WS-1
130 LET Y=Y+1: GO SUB 170
140 LET Y=Y-2: GO SUB 170
150 LET Y=Y+1: LET X=X-1: GO SUB 170
160 LET X=X+2: GO TO 110
170 IF POINT (X,Y)=1 THEN RETURN
180 PLOT X,Y: LET WS=WS+1
190 LET S(WS,1)=X: LET S(WS,2)=Y
200 RETURN
```

Instrukcje 10—40 demonstrują pracę właściwego podprogramu zamalowującego, który rozpoczyna się od linii 100. Przed wywołaniem podprogramu trzeba zmiennym **X** i **Y** nadać współrzędne dowolnego punktu leżącego wewnątrz konturu. Na początku deklarowana jest tablica **S**, mieszcząca stos, a wskaźnik stosu **WS** jest zerowany (stos jest na razie pusty). Jak wynika z deklaracji, liczba warstw stosu nie może przekroczyć 300. Podprogram 170 podejmuje próbę zapalenia punktu o współrzędnych **X**, **Y**. Jeśli punkt jest już zapalony, następuje natychmiastowy powrót, w przeciwnym razie punkt jest zapalany, a jego współrzędne — układane na stosie (linia 180—190). Po obsłużeniu sąsiadów ze stosu pobierane są współrzędne punktu znajdującego się na szczycie (linia 120), po czym następują próby zapalenia wszystkich czterech sąsiadów (linie 130—160).

Zauważmy, że gdyby zmienić warunek w linii 170, porównując wartość funkcji **POINT** nie z 1, a z Ø, oraz zmienić instrukcję **PLOT** na **PLOT INVERSE 1**, otrzymalibyśmy program, który ściera z ekranu dowolną wypełnioną figurę. Ten wariant pracy lepiej przypomina pożar kartki. Oczywiście w chwili wywołania zmienne **X** i **Y** musiałyby wskazywać na dowolny wypełniony punkt wnętrza figury.
Prostota naszego programu jest okupiona jego wielkim "apetytem" na pamięć. Wysokość stosu rośnie bardzo szybko (przez wysokość stosu rozumiemy liczbę ułożonych na nim elementów). Już przy stosunkowo niewielkich figurach może nastąpić przepełnienie tablicy **S**. Powiększenie rozmiaru tablicy **S** w deklaracji jest rozwiązaniem niezbyt skutecznym, gdyż ogranicza z kolei pamięć dostępną dla programu i danych. Znacznie rozsądniej jest zorganizować stos nie za pomocą tablicy numerycznej, lecz tekstowej. Żadna ze współrzędnych ekranowych nie przekracza 255. Współrzędne można więc pomieścić w pojedynczych bajtach. Jedyną niewygodą jest konieczność zapisu danych do tablicy funkcją **CHR$**, a odczytu przez **CODE**. Rozwiązanie takie zastosowano w przedstawionym niżej podprogramie zamalowującym **PĘDZEL**. Pozwala ono pięciokrotnie zwiększyć dopuszczalną wysokość stosu przy nie zmienionym w stosunku do programu poprzedniego zapotrzebowaniu na pamięć dla mieszczącej stos tablicy:

Program 72

```
10 CIRCLE 128,70,20: CIRCLE 126,72,14
20 LET X=112: LET Y=71: LET RP=0
30 GO SUB 9770: PRINT "WYPELNIONO: ";IP
40 LET X=112: LET Y=71: LET RP=1
50 GO SUB 9770: PRINT "SKASOWANO:  ";IP
60 STOP

9770 REM PEDZEL(X,Y,RP)
9771 DIM S$(1500,2): LET WS=0: LET IP=0
9772 GO SUB 9780: IF WS=0 THEN RETURN
9773 LET X=CODE S$(WS,1)
9774 LET Y=CODE S$(WS,2)
9775 LET WS=WS-1
9776 LET Y=Y+1: GO SUB 9780
9777 LET Y=Y-2: GO SUB 9780
9778 LET Y=Y+1: LET X=X-1: GO SUB 9780
9779 LET X=X+2: GO TO 9772
9780 IF X<0 OR X>255 THEN RETURN
9781 IF Y<0 OR Y>175 THEN RETURN
9782 IF POINT (X,Y)=RP=0 THEN RETURN
9783 PLOT INVERSE RP;X,Y
```

```
9784 LET WS=WS+1: LET IP=IP+1
9785 LET S$(WS,1)=CHR$ X
9786 LET S$(WS,2)=CHR$ Y
9787 RETURN
```

Krótki program pokazowy (linie 1Ø—6Ø) ukazuje możliwości podprogramu **PĘDZEL**. Są one większe niż w poprzednim przykładzie, gdyż podprogram zawiera kilka ulepszeń. Przede wszystkim granice ekranu interpretowane są jako element konturu: zapewniają to instrukcje 978Ø—9781. Oprócz tego podprogram może pracować w dwóch trybach: zamalowującym i ścierającym figurę. Wybór trybu odbywa się za pomocą parametru rodzaju pracy **RP**. Jeśli w chwili wywołania **RP**=Ø, to kontur będzie zamalowywany, a **RP**=1 oznacza, że program będzie starał się skasować z ekranu wypełnioną uprzednio figurę. W obydwu przypadkach zmienne **X** i **Y** powinny w chwili wywołania wskazywać punkt należący odpowiednio do wnętrza zamalowywanej lub ścieranej z ekranu figury.

Oprócz funkcji czysto graficznych podprogram **PĘDZEL** można z powodzeniem wykorzystać w charakterze ekranowego planimetru. Właściwość tę zawdzięcza **PĘDZEL** zmiennej **IP**. Zmienna ta jest zerowana w chwili wywołania, a potem zwiększa swą wartość o 1 przy każdym zapalonym lub skasowanym punkcie (linia 9784). W konsekwencji po powrocie z podprogramu zmienna **IP** przedstawia liczbę punktów wypełnionych lub skasowanych. W tym drugim przypadku zostaną uwzględnione także punkty konturowe. Nie należy się więc dziwić, że po wypełnieniu, a następnie skasowaniu określonej figury otrzymuje się różne wartości zmiennej **IP** (prog. 72, linie 3Ø i 5Ø). Ta druga wartość jest większa od pierwszej dokładnie o liczbę punktów tworzących kontur figury.

Mimo poprawnej pracy podprogramu **PĘDZEL**, w wielu zastosowaniach jego działanie z pewnością okaże się za wolne. Postęp w dziedzinie szybkości będzie możliwy po rezygnacji z języka BASIC i sięgnięciu po język maszynowy. Przedstawiony niżej program maszynowy realizuje te same funkcje, co podprogram **PĘDZEL**:

Program 73

```
9720 REM ZAMALOWYWANIE KONTURU
9721 DATA 50000,9721
9722 DATA "AF18023E0116001803Ø9"
9723 DATA "16FFAF2A8D5C228F5CE4"
9724 DATA "D95F01ØØØØD9ED4B65AF"
9725 DATA "5C21C0FFED42E5DDE10E"
9726 DATA "06FFC5ED4BB05CCDAA05"
9727 DATA "C3C1042823DDE5DD39AB"
9728 DATA "DDE1D2151F78FEB0DCC6"
```

```
9729 DATA "AAC33EFE8047DCAAC3B9"
9730 DATA "040CC4AAC33EFE814F4D"
9731 DATA "38D818D9D9C5D9C1C902"
9732 DATA "C5CDAA223C47A44F1EF2"
9733 DATA "01CB0B10FC7EAAA37927"
9734 DATA "C1C0D903A3D920057E7C"
9735 DATA "AAB3AA77D5CDDB08D1D7"
9736 DATA "E1C5E98F","""
```

Podprogram nie jest przesuwny i został zlokalizowany pod adresem 5ØØØØ. Jego długość wynosi 129 bajtów. Aby umożliwić Czytelnikom samodzielne zainstalowanie programu w dowolnym miejscu pamięci, opracowano specjalny sposób ładowania. Należy wykonać **RESTORE 972Ø**, wstawić w linii 9721 właściwy adres ładowania, po czym uruchomić poniższy program:

Program 74

```
200 GO SUB 9990: LET A=A-129
210 LET Q2=INT ((A+90)/256): LET Q1=A+90-256*Q2
220 POKE A+44,Q1: POKE A+45,Q2
230 POKE A+63,Q1: POKE A+64,Q2
240 POKE A+70,Q1: POKE A+71,Q2
250 POKE A+75,Q1: POKE A+76,Q2
```

Program maszynowy zostanie wpisany do pamięci począwszy od podanego adresu ładowania przez nasz standardowy loader (podprogram 999Ø). Następujące po wywołaniu loadera instrukcje **POKE** modyfikują kod programu maszynowego, przystosowując go do pracy w innym obszarze pamięci niż przewidziany pierwotnie. Jest rzeczą bardzo ważną, aby przed uruchomieniem loadera wykonać instrukcję **CLEAR** z argumentem przynajmniej o 1 mniejszym, niż wpisany w linii 9721 adres ładowania. Przedstawiony program maszynowy korzysta w trakcie pracy z tzw. systemowego stosu maszynowego. W razie jego przepełnienia nic nie grozi: praca programu zostanie przerwana, na ekranie zaś pojawi się komunikat "Out of memory" (brak wolnej pamięci). Zajdzie to jednak tylko wtedy, gdy wykonana została prawidłowo wspomniana instrukcja **CLEAR**. W przeciwnym razie grozi nie tylko zniszczenie samej procedury zamalowującej, ale i programu w języku BASIC.

Procedurę można wywołać w trojaki sposób. Wywołanie od adresu ładowania (**adrp**) powoduje zamalowywanie figury ciągłą plamą. Wywołanie od adresu **adrp+3** powoduje także zamalowywanie, ale nie plamą, lecz siatkowym wzorkiem. Wywołanie od adresu **adrp+9** powoduje kasowanie wypełnionej figury. Procedura wywoływana jest jako funkcja, może więc zwrócić

wartość. W przypadku wywołań od **adrp** i **adrp+9** wartością funkcji jest liczba wypełnionych lub skasowanych na ekranie punktów.
Jeśli korzystamy z zamalowywania wzorem, to mamy do wyboru dwa rodzaje siatki. Przełączenia dokonujemy wpisując do komórki **adrp+96** wartość 172, odtwarzamy zaś wzór pierwotny wpisując do tej samej komórki liczbę 164. Zwracana do programu wartość w przypadku zamalowywania wzorem nie odpowiada niestety liczbie zapalonych lub zgaszonych punktów, a zapotrzebowanie na pamięć jest większe niż przy kasowaniu obszaru lub zamalowywaniu go. Przekazywanie podprogramowi informacji o punkcie startu odbywa się przez wpisanie współrzędnej X do komórki 23728, a współrzędnej Y do komórki 23729. Obydwa bajty leżą wprawdzie w obszarze zmiennych systemowych, jednak normalnie są nie wykorzystane (były przeznaczone do przechowywania wektora przerwania niemaskowalnego NMI). Pozostałe wątpliwości co do sposobu korzystania z maszynowej procedury zamalowującej powinien rozwiać poniższy przykład:

Program 75

```
 10 CLEAR 49999: GO SUB 9990
 20 FOR R=1 TO 3
 30   CIRCLE 128,60+7*R,R*19
 40 NEXT R
 50 POKE 23728,128: POKE 23729,130
 60 RANDOMIZE USR 50000
 70 POKE 23729,110: RANDOMIZE USR 50003
 80 POKE 50096,172
 90 POKE 23729,70: RANDOMIZE USR 50003
100 CIRCLE 38,87,30: POKE 23728,40
110 PRINT "WYPELNIONO ";USR 50000;" PKT."
120 PAUSE 50
130 PRINT "SKASOWANO  ";USR 50009;" PKT."
140 STOP
```

Program maszynowy działa bardzo szybko i nie uszczupla pamięci programu i zmiennych języka BASIC.

Diagramy kołowe

Wyniki najbardziej złożonych obliczeń nie na wiele się zdadzą, jeżeli nie przedstawimy ich w przejrzystej i sugestywnej formie. Jest to szczególnie istotne wtedy, gdy trzeba dokonać szybkiej oceny i porównania kilku wielkości. Najwygodniej posłużyć się wtedy odpowiednim przedstawieniem graficznym. Bardzo czytelny jest diagram kołowy. Jego istota polega na tym, że każdej z porównywanych wielkości przyporządkowuje się wycinek koła

o powierzchni proporcjonalnej do jej wartości. Diagram przypomina wtedy tort podzielony na sektory różnej wielkości.

Załóżmy, że dysponujemy danymi przedstawiającymi liczbę kalorii zawartych w dziewięciu składnikach spożytych pewnego dnia posiłków. Zapisujemy te dane w postaci zbioru **DATA** i zamierzamy sporządzić z nich diagram kołowy w celu lepszego zorientowania się, które elementy naszego menu mają decydujący wpływ na konieczność stałego wydłużania paska u spodni. Niech zmienna **IW** przedstawia liczbę elementów wykresu (tu: 9). Wszystkie dane przenosimy ze zbioru **DATA** do tablicy **X**. Wystarczy posłużyć się pętlą **FOR... NEXT**.

Program 76a

```
  10 DIM X(30)
  20 LET IW=9
  30 FOR I=1 TO IW
  40   READ X(I)
  50 NEXT I

1000 REM DIAGRAM KOLOWY
1010 CIRCLE 92,84,83
1020 LET ALFA=0: LET S=0
1030 FOR I=1 TO IW
1040   LET S=S+X(I)
1050 NEXT I
1060 FOR I=1 TO IW
1070   LET ALFA=2*PI/S*X(I)+ALFA
1080   PLOT 92,84
1090   DRAW 83*COS ALFA,83*SIN ALFA
1100 NEXT I

2000 DATA 11,30,50,20,30,90,25,44,9
```

Przygotowanie diagramu rozpoczyna się w linii 1Ø1Ø wykreśleniem okręgu wyznaczającego obrys "tortu". Następnie obliczana jest suma wszystkich składników (zmienna **S**). Teraz można już rozpocząć podział "tortu". Kąt, pod jakim biegnie promień oddzielający kolejny sektor od reszty tarczy, wyznaczany jest w linii 1Ø7Ø. Kąt ten jest proporcjonalny do stosunku wartości danego elementu do sumy **S**. Po obliczeniu kąta wystarczy już wypełnić punkt pośrodku tarczy i poprowadzić od niego pod wyznaczonym kątem promień.

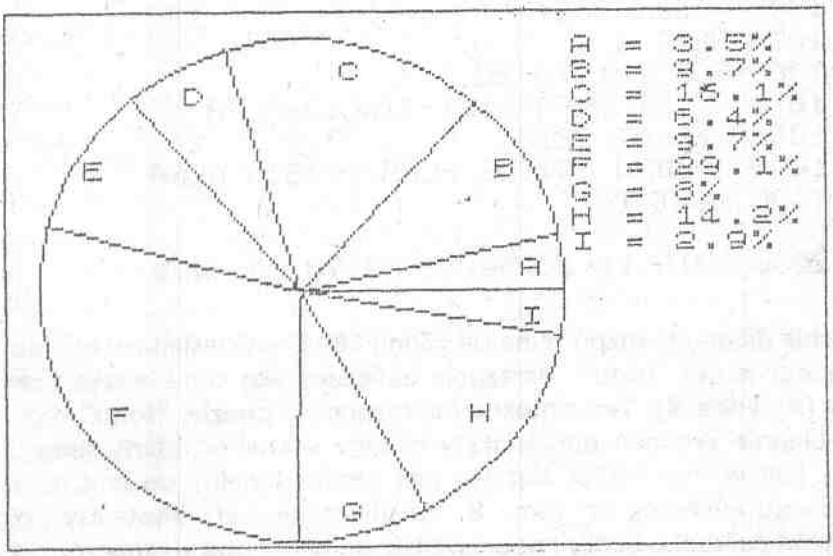
Sporządzony diagram należy opisać. Najprościej będzie każdy sektor oznaczyć kolejną literą alfabetu, poczynając od "A". Z prawej strony wyszczególnimy w postaci tablicy liczbowej procentowy udział każdego z sektorów w całości. Dopiszemy do naszego programu następujące instrukcje:

Program 76b

```
1065 LET ALFA0=ALFA

1091 LET P=(ALFA+ALFA0)/2
1092 PRINT AT 11-9*SIN P,11+9*COS P;CHR$ (64+I);
1093 PRINT AT I-1,23; BRIGHT 1;CHR$ (64+I);
1094 PRINT " = ";INT (1000*X(I)/8)/10;"%";
```

Wprowadzona zmienna pomocnicza **ALFA Ø** przechowuje kąt, pod którym biegł promień odgradzający poprzedni sektor. Zmienną tę wykorzystujemy w linii 1Ø91 do wyznaczenia położenia osi symetrii każdego sektora. Stanowi to podstawę do obliczenia miejsca, w którym należy wyświetlić odpowiadającą danemu sektorowi literę (linia 1Ø92).
Ponieważ wyświetlane dane mają i tak charakter wyłącznie orientacyjny, nie byłoby celowe podawać procentowych udziałów ze zbyt dużą liczbą miejsc po przecinku. W zupełności wystarczy jedna cyfra. Zapewnia to postać wyrażenia w linii 1Ø94. Przykład gotowego diagramu kołowego ukazuje rysunek 9. Diagramy kołowe są bardzo skutecznym sposobem prezentacji danych, jeżeli liczba elementów nie przekracza 1Ø÷13, a udział żadnego z nich nie jest mniejszy niż ok. 2%.



Rys. 9 Diagram kołowy uzyskany za pomocą programu 76

Wykresy funkcji jednej zmiennej

Możliwość automatycznego sporządzania wykresów zadanych funkcji jest nieraz bardzo cenna, gdyż ułatwia szybką ocenę charakteru funkcji. Wykres jest znacznie przejrzystszą formą przedstawiania zależności niż sam suchy wzór. W praktyce najczęściej korzystamy z funkcji jednej zmiennej postaci y=f(x), przedstawionych w postaci wykresu w układzie współrzędnych prostokątnych (kartezjańskim).

Zasada komputerowego kreślenia funkcji nie różni się od kreślenia ręcznego. Należy zakres zmian zmiennej niezależnej (argumentu funkcji) podzielić na określoną liczbę przedziałów, a następnie wyliczyć wartość funkcji w każdym z punktów granicznych i nanieść punkt na papier lub ekran.

Dobór skali osi X najlepiej powierzyć komputerowi, tak aby przedział zmian argumentu był na ekranie jak najbardziej rozciągnięty.

Jeśli chodzi o dobór skali osi Y, możliwe są dwa podejścia. W pierwszym z nich komputer bada funkcję w całym przedziale zmian zmiennej niezależnej jeszcze przed właściwym kreśleniem wykresu, znajdując wartości minimalne i maksymalne. Na tej podstawie dobierana jest skala i przesunięcie na osi Y tak, aby wykres wypełnił cały ekran. Rozwiązanie takie jest wygodne, lecz nie pozwala na dokładniejsze obejrzenie szczególnie interesujących fragmentów wykresu ani na taki wybór obserwowanego okna, aby na ekranie widać było np. także i osie układu współrzędnych.

Podejście drugie zakłada, że użytkownik poda dopuszczalny zakres zmian zmiennej Y, definiując niejako współrzędną Y górnej i dolnej krawędzi ekranu. Większa swoboda w wyborze postaci wykresu okupiona jest tutaj koniecznością wstępnego oszacowania możliwych wartości funkcji. Największym problemem jest możliwość wystąpienia takich wartości funkcji, których przedstawienie w zdefiniowanym na ekranie oknie nie będzie już możliwe i doprowadzi do błędu w trakcie wykonywania instrukcji **PLOT** lub **DRAW** i w konsekwencji do przerwania pracy programu. Jeszcze gorsza sytuacja wystąpi podczas wykreślania funkcji mających w przedziale zmian argumentu punkty nieciągłości (np. funkcja Y=1/X dla X z przedziału (−1, 1)). Programy z automatycznym skalowaniem osi Y zawiodą tu zupełnie.

Przedstawiony poniżej prosty program pozwala na uzyskanie wykresu praktycznie dowolnej funkcji postaci y=f(x), umożliwiając uniknięcie problemów z punktami nieciągłości i fragmentami wykresu wybiegającymi poza ekran:

Program 77

```
10 REM Wykres funkcji Y=f(X)
20 INPUT "Najmniejsza wartosc X: ";Xmin
30 INPUT "Najwieksza  wartosc X: ";Xmax
40 INPUT "Najmniejsza wartosc Y: ";Ymin
50 INPUT "Najwieksza  wartosc Y: ";Ymax
```

```
 60 LET DeltaX=(Xmax-Xmin)/256
 70 LET SkalaX=255/(Xmax-Xmin)
 80 LET SkalaY=175/(Ymax-Ymin)
 90 CLS
100 LET X=-SkalaX*Xmin
110 LET Y=-SkalaY*Ymin
120 IF X>=0 AND X<=255 THEN PLOT X,0: DRAW 0,175
130 IF Y>=0 AND Y<=175 THEN PLOT 0,Y: DRAW 255,0
140 LET W1=1: LET W2=0
150 FOR X=xmin TO Xmax STEP DeltaX
160 LET X2=INT (SkalaX*(X-Xmin)+.5)
170 LET W2=FN B(X)
180 IF W2=0 THEN LET Y=FN Y(X)
190 IF Y<Ymin OR Y>Ymax THEN LET W2=1
200 IF W2<>0 THEN LET W1=1: GO TO 260
210 IF W1=1 THEN GO TO 250
220 LET Y2=INT (SkalaY*(Y-Ymin)+.5)
230 PLOT X1,Y1: DRAW X2-X1,Y2-Y1
240 LET Y1=Y2: GO TO 260
250 LET Y1=INT (SkalaY*(Y-Ymin)+.5): LET W1=0
260 LET X1=X2
270 NEXT X

3000 DEF FN Y(X)=(2-X)/((X-1)*(X-3))
3010 DEF FN B(X)=(X=1) OR (X=3)
```

Na początku program pyta o dopuszczalny zakres zmian zmiennych X i Y. Następnie wyznaczany jest elementarny przyrost zmiennej X (**DeltaX**) tak, aby zakres zmian argumentu podzielić na 255 przedziałów. Współczynniki skali dla obydwu osi (**SkalaX** i **SkalaY**) dobierane są tak, by wykorzystać całą powierzchnię okna użytkownika.

Po skasowaniu ekranu (linia 90) program wyznacza współrzędne ekranowe obydwu osi układu. Jeśli okaże się, że dana oś mieści się w oknie, jest wykreślana (linie 120 i 130). W linii 150 rozpoczyna pracę właściwa pętla obliczająca kolejne wartości funkcji i dorysowująca fragmenty wykresu. Postać wykreślanej funkcji zdefiniowana jest w linii 3000 i może być oczywiście dowolnie zmieniana przez użytkownika (możliwe jest także wczytywanie wzoru funkcyjnego jako zmiennej łańcuchowej, a następnie obliczanie jego wartości funkcją **VAL**).

Funkcja **FN B** w linii 3010 spełnia istotną rolę w przypadku funkcji z punktami nieciągłości. Postać tej funkcji trzeba dobrać tak, aby we wszystkich punktach nieciągłości **FN B** była różna od 0 (**FN Y** w programie 77 ma dwa punkty nieciągłości: dla X=1 i X=3). Tam, gdzie funkcja jest ciągła w całym przedziale zmian argumentu, wystarczy podstawić:

DEF FN B(X)=0

Wartość **FN B** jest wyznaczana jeszcze przed obliczaniem wartości **FN Y** i nie dopuszcza do prób wyliczania funkcji w punktach nieciągłości, co prowadziłoby do błędów. Wskaźnikiem jest zmienna **W2**. Po wyznaczeniu wartości funkcji następuje sprawdzenie, czy odpowiedni punkt mieści się w zdefiniowanym oknie (linia 19Ø). Jeśli nie, punkt nie jest wykreślany, a zmienna **W2** przyjmuje wartość różną od Ø. Wartość **W2** jest więc zerem tylko wtedy, gdy wartość funkcji została poprawnie wyliczona, a punkt mieści się w oknie. Zmienna **W1** pozwala rozpoznawać sytuacje, w których nie wolno łączyć punktu, mieszczącego się w oknie, z punktem poprzednim. Jeśli **W1**=1, znaczy to, że poprzedni punkt leżał poza oknem i nie wolno łączyć go z następnikiem przy użyciu instrukcji **DRAW**. Tylko w sytuacji, gdy zarówno **W1**, jak i **W2** są równe Ø (punkt poprzedni i punkt aktualny mieściły się w oknie ekranu), może nastąpić połączenie tych punktów linią ciągłą (wiersze 22Ø—24Ø). Przed rozpoczęciem kreślenia wykresu, a więc przed obliczeniem współrzędnych pierwszego punktu, zmienna **W1** ustawiana jest na 1. Symuluje to sytuację, w której poprzedni punkt leżał poza ekranem, a w rzeczywistości zapobiega łączeniu pierwszego punktu wykresu z nie istniejącym poprzednikiem.

Elementy grafiki pseudoprzestrzennej

Trójwymiarowa grafika na ekranie komputera wzbudza często zachwyt widzów. Z praktycznego punktu widzenia pseudoprzestrzenne przedstawienie różnych obiektów: brył budynków, części maszyn, pojazdów, wykresów funkcji wielu zmiennych itp. znacznie zwiększa przejrzystość i lepiej przemawia do wyobraźni.

Aby przedstawić trójwymiarowy obiekt na płaszczyźnie ekranu, trzeba dokonać odpowiedniego rzutowania (projekcji). Rozróżnia się zasadniczo dwa rodzaje projekcji perspektywy: równoległą (aksonometryczną) i centralną (perspektywę rzutu środkowego). Ta ostatnia lepiej odpowiada naturalnemu sposobowi widzenia i daje wierniejsze odwzorowanie, wymaga jednak bardziej złożonych obliczeń i wyboru warunków początkowych, np. położenia wyimaginowanego obserwatora. Zajmiemy się więc tylko mniej skomplikowaną projekcją równoległą.

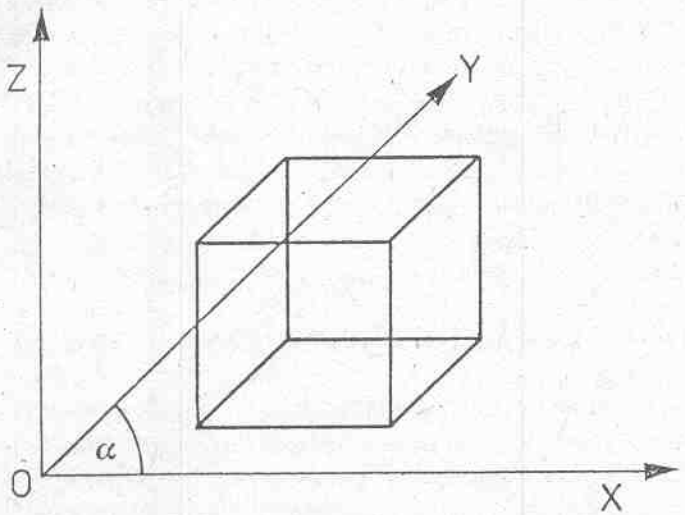
Projekcja aksonometryczna przedstawia bryły w ten sposób, że linie równoległe w przestrzeni zostają równoległe także na płaszczyźnie rzutu. Proporcje odcinków równoległych do płaszczyzny wyznaczonej osiami X i Z są zachowane, długości pozostałych odcinków zostają skrócone w zależności od kąta nachylenia każdego z nich do płaszczyzny rzutu (rys. 10).

W trójwymiarowym układzie współrzędnych przestrzennych każdy punkt bryły opisany jest współrzędnymi X, Y, Z. Podczas rzutowania na płaszczyznę każda z tych trójek musi zostać sprowadzona do pary współrzędnych ekranowych: Xe, Ye.

Przyjmując, że odcinki równoległe do osi Y ulegają skróceniu o ustalony współczynnik skrócenia K i że znany jest kąt α, stwierdzimy, że całe przekształcenie jest jednoznacznie określane przez K i α i że można je sprowadzić do następujących wzorów na współrzędne ekranowe:

$$Xe = X + Y * K * \cos\alpha + XeØ$$
$$Ye = Z + Y * K * \sin\alpha + YeØ$$

XeØ i **YeØ** są tutaj stałymi, wyznaczającymi położenie w płaszczyźnie ekranu rzutu środka przestrzennego układu współrzędnych XYZ.



Rys. 10 Rzut sześcianu na płaszczyznę w projekcji równoległej

Powyższe wzory będą punktem wyjścia do skonstruowania prostego programu, wyświetlającego na ekranie rzut aksonometryczny dowolnej konstrukcji trójwymiarowej, złożonej z wierzchołków i łączących je krawędzi:

Program 78a

```
 1 REM Grafika trojwymiarowa
10 DIM X(100): DIM Y(100): DIM Z(100)
20 LET K=0.6: LET ALFA=0.7
30 LET SK=7
40 LET KX=K*COS ALFA
50 LET KY=K*SIN ALFA
60 LET N=0
70 READ Q: IF Q>=1E37 THEN GO TO 100
80 LET N=N+1: LET X(N)=Q: READ Y(N),Z(N)
90 GO TO 70
```

```
 100 CLS
 110 READ A: IF A=0 THEN GO TO 200
 120 READ B: LET Xe=FN X(A): LET Ye=FN Y(A)
 130 PLOT Xe,Ye: DRAW FN X(B)-Xe,FN Y(B)-Ye
 140 GO TO 110

4800 DEF FN X(A)=SK*(X(A)+KX*Y(A))+128
4900 DEF FN Y(A)=SK*(Z(A)+KY*Y(A))+87

5000 DATA -5,-5,-5, -5,5,-5, 5,5,-5, 5,-5,-5
5010 DATA -5,-5,5, -5,5,5, 5,5,5, 5,-5,5
5020 DATA 1E37
6000 DATA 1,2, 2,3, 3,4, 4,1, 5,6, 6,7, 7,8, 8,5
6010 DATA 1,5, 2,6, 3,7, 4,8
6020 DATA 1,6, 2,7, 2,5
6030 DATA 0
```

Przekształcenie współrzędnych przestrzennych XYZ na współrzędne ekranowe zrealizujemy za pomocą zdefiniowanych funkcji **FN Y** i **FN X**. Argumentem funkcji jest numer wierzchołka, którego współrzędne rzutu aktualnie wyznaczamy. Zmienna **SK** jest współczynnikiem skali, pozwalającym na zmniejszanie lub zwiększanie rzutu bryły w miarę potrzeby. Zarówno **SK**, jak i współczynnik skrótu **K** oraz kąt **ALFA** można modyfikować, wybierając najkorzystniejsze dla danej bryły warunki projekcji.

Informacja o samej bryle zawarta jest w dwóch blokach **DATA**. Pierwszy blok, rozpoczynający się od linii 5ØØØ, zawiera współrzędne poszczególnych wierzchołków. Przyjmujemy, że wszystkie wierzchołki bryły są jednoznacznie ponumerowane, poczynając od 1. Współrzędne wierzchołków muszą być zapisane w bloku **DATA** w kolejności numeracji. Wierzchołek nr 1 ma współrzędne (−5,−5,−5), wierzchołek nr 4: (5,−5,−5) itd. Blok danych opisujących wierzchołki zakończony jest pojedynczą liczbą nie mniejszą od $1Ø^{37}$ (linia 5Ø2Ø). Liczba ta spełnia funkcję "wartownika": jej odczytanie oznacza, że nie ma więcej wierzchołków.

Współrzędne wierzchołków są na początku pracy programu odczytywane ze zbioru **DATA** i przenoszone do trzech tablic **X**,**Y**,**Z**. Indeks danej współrzędnej w tablicy odpowiada numerowi wierzchołka, który opisuje. Przy okazji wpisywania współrzędnych do tablic zliczana jest liczba wierzchołków (zmienna **N**).

Wykreślanie rzutu bryły rozpoczyna się w linii 1ØØ skasowaniem ekranu. Rysowane będą jednak nie same wierzchołki, lecz łączące je krawędzie. Potrzebujemy dodatkowych informacji o tym, które wierzchołki mają być połączone odcinkami. Informacje te zawarte są w drugim bloku **DATA**, od linii 6ØØØ. Występują tu pary liczb, określających wierzchołki do połączenia. Każda para definiuje jedną krawędź. Np. para **1, 2** oznacza, że trzeba poprowadzić linię od wierzchołka nr 1 do wierzchołka nr 2 itd. Zgromadzone

w naszym programie przykładowym dane opisują sześcian, w którym dodatkowo w jednej ze ścian poprowadzono dwie przekątne, a w drugiej — jedną. Blok opisu krawędzi także kończy się "wartownikiem". Tym razem jest to Ø — wierzchołka o takim numerze być nie może.
Po odczytaniu obu numerów wierzchołków wyznaczane są współrzędne ekranowe pierwszego (linia 12Ø), a następnie prowadzony jest odcinek od pierwszego do drugiego (linia 13Ø). Transformację współrzędnych wykonują funkcje **FN X** i **FN Y**. Odpowiadają one przytoczonym wzorom. Stałe 128 i 87 to miejsce rzutu środka układu współrzędnych. W razie potrzeby można je zmienić odpowiednio do potrzeb.
Chcąc wykreślić rzut innej bryły, wystarczy wymienić zawartość obydwu bloków **DATA**. Wygodnie jest naszkicować bryłę na kartce papieru, ponumerować wierzchołki, a dopiero potem wprowadzać dane. Można też pokusić się o sporządzenie programu, pozwalającego szkicować wprost na ekranie. Jest to nieco bardziej złożone.
Po sporządzeniu nowych bloków danych nie należy zapomnieć o zamknięciu każdego z nich właściwym wskaźnikiem końca — "wartownikiem".
Nasz program sprawnie wykreśla kontury brył o znanych współrzędnych wierzchołków, lecz poza tym jego możliwości są skromne. Można co najwyżej eksperymentować ze zmianą skali i współczynnikami projekcji. Chcielibyśmy jednak móc dokonywać także dowolnych obrotów bryły wokół każdej z trzech osi układu współrzędnych w celu obserwacji jej z różnych kierunków. Także i tu niezbędna będzie odpowiednia transformacja współrzędnych przestrzennych. Załóżmy, że chcemy obrócić bryłę wokół osi X o kąt βx. Współrzędne X poszczególnych wierzchołków pozostaną nie zmienione, a nowe współrzędne Y' i Z' można wyznaczyć z zależności:

$$Y' = Y * \cos\beta x + Z * \sin\beta x$$
$$Z' = -Y * \sin\beta x + Z * \cos\beta x$$

Wzory te można znaleźć w podręcznikach geometrii lub pokusić się o ich samodzielne wyprowadzenie. Wystarczy sporządzić prosty szkic sytuacyjny. W przypadku obrotu bryły wokół osi Y wzory transformacyjne będą podobne:

$$X' = X * \cos\beta y + Z * \sin\beta y$$
$$Y' = Y$$
$$Z' = -X * \sin\beta y + Z * \cos\beta y$$

Analogicznie przy obrocie wokół osi Z:

$$X' = X * \cos\beta z + Y * \sin\beta z$$
$$Y' = -X * \sin\beta z + Y * \cos\beta z$$
$$Z' = Z$$

Dowolny obrót bryły w przestrzeni można złożyć z trzech niezależnych obrotów względem każdej z osi. Nie pozostaje nam nic innego, jak uzupełnić nasz program 78a o możliwość obracania zdefiniowanej bryły. Po wykreśle-

niu rzutu program pyta nas o kąty obrotu bryły wokół każdej z trzech osi układu współrzędnych:

Program 78b

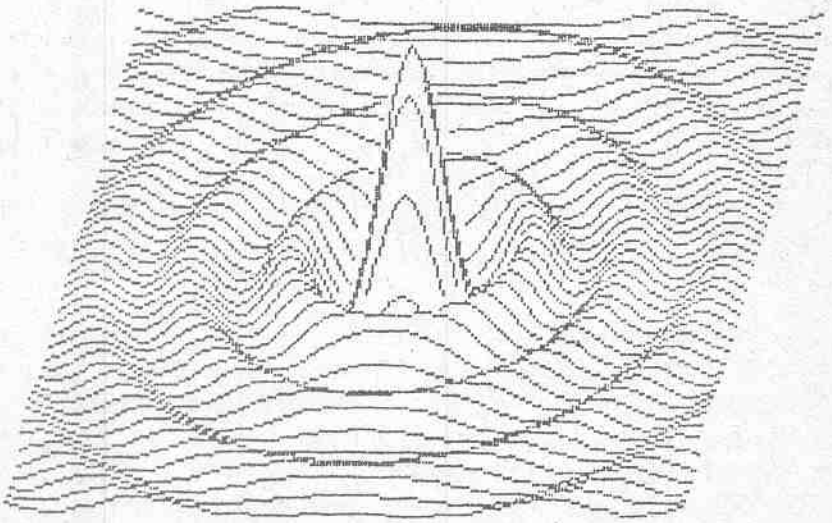
```
200 INPUT "Kat obrotu wokol osi X: ";BX
210 INPUT "Kat obrotu wokol osi Y: ";BY
220 INPUT "Kat obrotu wokol osi Z: ";BZ
230 LET SX=SIN (PI*BX/180): LET CX=COS (PI*BX/180)
240 LET SY=SIN (PI*BY/180): LET CY=COS (PI*BY/180)
250 LET SZ=SIN (PI*BZ/180): LET CZ=COS (PI*BZ/180)
260 FOR I=1 TO N
270  LET Y1=Y(I): LET Z1=Z(I)
280  LET Y(I)=Y1*CX+Z1*SX: LET Z(I)=-Y1*SX+Z1*CX
290  LET X1=X(I): LET Z1=Z(I)
300  LET X(I)=X1*CY+Z1*SY: LET Z(I)=-X1*SY+Z1*CY
310  LET X1=X(I): LET Y1=Y(I)
320  LET X(I)=X1*CZ+Y1*SZ: LET Y(I)=-X1*SZ+Y1*CZ
330 NEXT I
340 RESTORE 6000: GO TO 100
```

Program pozwala podawać poszczególne kąty w mierze stopniowej. Ponieważ wartości poszczególnych sinusów i kosinusów potrzebne będą wielokrotnie, a obliczanie wartości funkcji trygonometrycznych jest czasochłonne, wartości poszczególnych funkcji zostaną obliczone tylko raz i zapamiętane jako gotowe współczynniki transformacji (np. **SX** = sinβx, a **CZ** = cosβz). W trakcie obliczania współczynników miara kątowa przeliczana jest na wymaganą łukową.

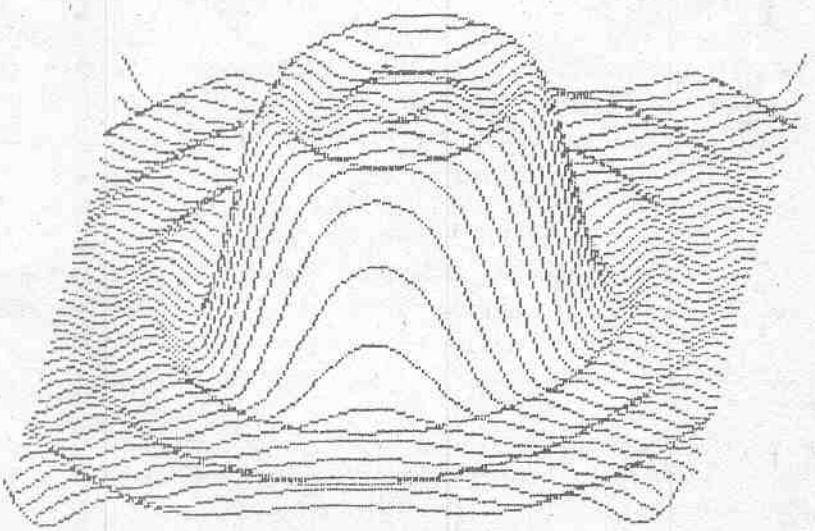
Teraz można przystąpić do przekształcania współrzędnych, czyli właściwego obrotu bryły. Dla prostoty wykonamy kolejno trzy obroty: wokół osi X (linie 270 i 280), osi Y (linie 290 i 300) oraz osi Z (linie 310 i 320). Obliczenia odbywają się na podstawie przedstawionych poprzednio wzorów. Operacja powtarza się oddzielnie dla każdego wierzchołka; po jej zakończeniu można wrócić do kreślenia rzutu bryły na ekranie. Najpierw trzeba jednak przesunąć wskaźnik odczytu zbioru **DATA** na początek bloku opisującego przebieg krawędzi (instrukcja **RESTORE 6000** w linii 340). Istnieje możliwość dokonania obrotu bryły "za jednym zamachem", w pojedynczej transformacji, bez rozbijania operacji na kolejne obroty wokół osi. Wzory komplikują się wtedy i stają mniej przejrzyste, albo też trzeba posłużyć się rachunkiem macierzowym.

Przygotowując dane bryły, którą zamierzamy obracać na ekranie, wskazane jest tak wybierać współrzędne wierzchołków, aby środek układu współrzędnych wypadł w środku bryły. Uchroni to przed wypadnięciem bryły poza ekran podczas obrotów. Lepszym, lecz bardziej złożonym rozwiązaniem, jest napisanie odpowiedniego programu, automatycznie centrującego bryłę.

Innym, popularnym zastosowaniem grafiki pseudoprzestrzennej jest rysowanie wykresów funkcji dwóch zmiennych. Zasada jest prosta: założony przedział zmienności obydwu argumentów (np. X i Y) dzieli się na odcinki równej długości, po czym wyznacza wartości funkcji dla wszystkich kombinacji X i Y. Obliczone wartości traktuje się jako współrzędną Z i przedstawia w poprzednio opisany już sposób. Dla zwiększenia przejrzystości sąsiednie punkty łączy się liniami w jednym, albo nawet w obydwu kierunkach (rys. 11a i b). W pierwszym przypadku wykres wypadkowy jest jakby złożeniem rodziny wykresów wykonanych dla zmiennej niezależnej X i różnych, ustalonych wartości zmiennej Y.



a)



b)

Rys. 11a, b Przestrzenne wykresy funkcji

Stosunkowo najtrudniejszy jest problem eliminacji niewidocznych linii. Niezbyt eleganckie, lecz proste rozwiązanie polega na zastosowaniu tablicy jednowymiarowej z 256 elementów: każdemu z nich przyporządkowujemy jedną z 256 możliwych współrzędnych osi Xe. Wykres trzeba kreślić od najmniejszych wartości Y do największych. Po wyznaczeniu kolejnej wartości Z należy obliczyć współrzędne ekranowe Xe i Ye. Następnie porównujemy Ye z tą komórką tablicy, która odpowiada części całkowitej Xe. Jeśli Ye jest większe, punkt rysujemy, a Ye wpisujemy do tablicy w miejsce poprzedniej zawartości. W przeciwnym razie punkt ignorujemy. Takie postępowanie opiera się na założeniu, że na dalszych planach widoczne są tylko punkty nie przysłonięte przez obiekty na planach bliższych i że powierzchnia wykresu jest ciągła. Ogólnie zagadnienie eliminacji ukrytych linii i wykresów funkcji wielu zmiennych jest złożone. Aby umożliwić dociekliwym Czytelnikom zgłębienie tych problemów, przytaczamy program, za pomocą którego sporządzono rys. 11a. Tablica **H** jest właśnie wspomnianą tablicą przesłaniania.

Program 79

```
2000 REM Przestrzenny wykres funkcji
2010 LET a=1080: LET k1=80
2020 LET w=75: LET k=.75
2030 LET Xe0=128: LET Ye0=83: LET bm=PI/180
2040 LET c=k*COS (w*bm): LET s=k*SIN (w*bm)
2050 LET dx=3: LET dy=5: LET af=a/90
2060 DIM H(256)
2070 FOR l=1 TO 256: LET H(l)=-1000: NEXT l
2080 FOR q=-110 TO 110 STEP dy: LET y=q*af
2090  FOR m=-105 TO 105 STEP dx
2100   LET x=m*af: GO SUB 2400
2110   LET Xe=INT (Xe0+m+c*q+.5): LET Ye=INT (Ye0+s*q+z+.5)
2130   IF m>-105 THEN GO TO 2170
2140   LET f1=0: LET l=INT (Xe/dx)
2150   IF Ye>=H(l+1) THEN LET f1=1: LET H(l+1)=Ye
2160   LET x1=Xe: LET y1=Ye: GO TO 2220
2170   LET f2=0: LET l=INT (Xe/dx)
2180   IF Ye>=H(l+1) THEN LET f2=1: LET H(l+1)=Ye
2190   LET x2=Xe: LET y2=Ye
2200   IF f1*f2=1 THEN PLOT x1,y1: DRAW x2-x1,y2-y1
2210   LET x1=x2: LET y1=y2: LET f1=f2
2220  NEXT m
2230 NEXT q
2250 STOP

2400 LET r=SQR (x*x+y*y)*bm
2410 IF r=0 THEN LET z=k1: RETURN
2420 LET z=k1*SIN (r)/r: RETURN
```

Aby otrzymać wykres z rys. 11b, trzeba do programu 79 wprowadzić następujące zmiany:

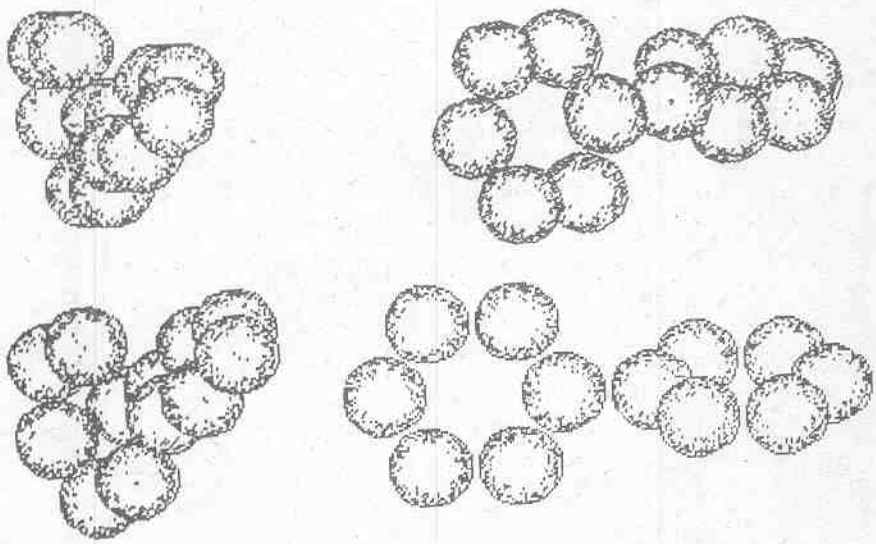
```
2010 LET a=155: LET k1=39
2020 LET w=75: LET k=.65

2400 LET r=SQR (x*x+y*y)*bm
2410 LET z=k1*(COS r-COS (3*r)/3+COS (5*r)/5-COS (7*r)/7)+24
2420 RETURN
```

Wartość funkcji obliczana jest w linii 2400 i następnych. Z uwagi na złożony charakter funkcji zrezygnowano z jednolitego zapisu funkcji.

Przestrzenne modele cząsteczek związków chemicznych

Interesującym i efektownym zastosowaniem komputerowej grafiki pseudoprzestrzennej jest przedstawianie na ekranie modeli cząsteczek związków chemicznych. Poszczególne atomy molekuły reprezentowane są przez kule. Średnica i sposób cieniowania kul mogą być zróżnicowane w zależności od rodzaju atomu. Cząsteczkę można poddawać na ekranie dowolnym obrotom, obserwując jej strukturę z różnych kierunków. Oprócz przedstawiania molekuł o znanej budowie można oczywiście także zabawiać się w modelowanie budowy cząsteczek o strukturze jeszcze niezupełnie poznanej (rys. 12).



Rys. 12 Model cząsteczki bisphenolu obserwowany pod różnymi kątami

Ponieważ nie zależy nam szczególnie na przedstawieniu cząsteczki w konkretnym układzie współrzędnych, możemy bardzo uprościć projekcję równoległą. Przyjmijmy, że nasz punkt widzenia leży na przedłużeniu osi Y (rys. 10). W tym przypadku współczynnik skrócenia zeruje się, co bardzo uprości funkcje transformacji współrzędnych przestrzennych na ekranowe. Przy przestrzennych obrotach cząsteczki wykorzystamy przekształcenia użyte już w programie 78b.

Do odtworzenia geometrii cząsteczki potrzebne będą nam wyłącznie współrzędne poszczególnych atomów, wyznaczone w jakimś — dowolnym — układzie współrzędnych prostokątnych i w jakiejś skali. Nasz model na ekranie powinien jednak oddawać nie tylko geometryczne zależności w cząsteczce, lecz także ułatwiać rozróżnienie poszczególnych rodzajów atomów. Można to osiągnąć przedstawiając atomy różnych pierwiastków za pomocą kul o różnych średnicach, stosując zróżnicowane cieniowanie kul itp. Zdecydujmy się na promień. Każdy atom będzie więc w naszym modelu reprezentowany przez czwórkę liczb: trzy współrzędne X, Y, Z oraz promień kuli, reprezentującej dany atom.

O ile samo rzutowanie cząsteczki na płaszczyznę ekranu nie sprawi nam kłopotu, o tyle wystąpi problem maskowania niewidocznych fragmentów kul przysłoniętych częściowo lub całkowicie przez inne kule, położone bliżej obserwatora. Rozwiązanie tego problemu jest niezbędne dla zapewnienia realistycznego, perspektywicznego obrazu cząsteczki. Trzeba zastanowić się też nad takim sposobem odwzorowania kul na płaszczyźnie, aby obserwator wyczuwał ich obły kształt. Osiągnąć można to przez odpowiednie cieniowanie.

Przykładem programu w języku BASIC, który w dostatecznym stopniu spełnia postawione wymagania, jest program 80. Odnajdziemy w nim wiele elementów poznanych już wcześniej.

Dane o strukturze molekuły zgrupowane są w postaci czwórek liczb w bloku **DATA** poczynając od linii 5ØØØ. Blok kończy się "wartownikiem" 1E37. Funkcje transformujące współrzędne przestrzenne na ekranowe uległy znacznemu uproszczeniu w związku z naszym założeniem o lokalizacji obserwatora (linia 48ØØ).

Trzy tablice dla współrzędnych zostały uzupełnione czwartą, dla promienia kuli — **R** (linia 1Ø). Do współczynnika skali **SK**, stosowanego do współrzędnych, doszedł drugi: **SR**. Określa on skalę, w jakiej rysowane są kule-atomy. Pętla w liniach 4Ø—7Ø przenosi dane z bloku **DATA** do tablic, zliczając równocześnie atomy (zmienna **N**).

W programie 78, wyświetlającym strukturę złożoną z wierzchołków i krawędzi, kolejność rysowania była obojętna. Tym razem będzie inaczej.

Program 80

```
 10 REM Przestrzenne modele czasteczek
 20 DIM X(100): DIM Y(100): DIM Z(100): DIM R(100)
 30 LET SK=25: LET SR=12
 40 LET N=0
 50 READ Q: IF Q>=1E37 THEN GO TO 100
 60 LET N=N+1: LET X(N)=Q: READ Y(N),Z(N),R(N)
 70 GO TO 50

100 CLS
110 LET F=0
120 FOR I=1 TO N-1
130  IF Y(I)>=Y(I+1) THEN GO TO 190
140  LET T=X(I): LET X(I)=X(I+1): LET X(I+1)=T
150  LET T=Y(I): LET Y(I)=Y(I+1): LET Y(I+1)=T
160  LET T=Z(I): LET Z(I)=Z(I+1): LET Z(I+1)=T
170  LET T=R(I): LET R(I)=R(I+1): LET R(I+1)=T
180  LET F=1
190 NEXT I: IF F=1 THEN GO TO 110

200 FOR I=1 TO N
210  LET Xe=FN X(I): LET Ye=FN Y(I): LET Re=SR*R(I)
220  GO SUB 1000
230 NEXT I

240 INPUT "Kat obrotu wokol osi X: ";BX
250 INPUT "Kat obrotu wokol osi Y: ";BY
260 INPUT "Kat obrotu wokol osi Z: ";BZ
270 LET SX=SIN (PI*BX/180): LET CX=COS (PI*BX/180)
280 LET SY=SIN (PI*BY/180): LET CY=COS (PI*BY/180)
290 LET SZ=SIN (PI*BZ/180): LET CZ=COS (PI*BZ/180)
300 FOR I=1 TO N
310  LET Y1=Y(I): LET Z1=Z(I)
320  LET Y(I)=Y1*CX+Z1*SX: LET Z(I)=-Y1*SX+Z1*CX
330  LET X1=X(I): LET Z1=Z(I)
340  LET X(I)=X1*CY+Z1*SY: LET Z(I)=-X1*SY+Z1*CY
350  LET X1=X(I): LET Y1=Y(I)
360  LET X(I)=X1*CZ+Y1*SZ: LET Y(I)=-X1*SZ+Y1*CZ
370 NEXT I
380 GO TO 100

1000 FOR J=0 TO Re-1 STEP .5: CIRCLE INVERSE 1;Xe,Ye,J
1010 NEXT J: CIRCLE Xe,Ye,Re
1020 FOR J=1 TO 150
1030  LET Fi=2*PI*RND
1040  LET D=Re*(1-RND*RND*0.4)-1
1050  LET A=D*SIN Fi: LET B=D*COS Fi
1060  PLOT Xe+A,Ye+B
1070 NEXT J
1080 RETURN
```

```
4800 DEF FN X(A)=SK*X(A)+128: DEF FN Y(A)=SK*Z(A)+87

5000 DATA -2.5,0,0,1, -2,0,0.87,1, -1,0,.87,1
5010 DATA -.5,0,0,1, -1,0,-.87,1, -2,0,-.87,1
5020 DATA .5,0,0,1, 1,.348,.348,1, 2,.348,.348,1
5030 DATA 2.5,0,0,1, 2,-.348,-.348,1, 1,-.348,-.348,1
5040 DATA 1E37
```

Zamierzamy przedstawić zbiór kul, umieszczonych w różnej odległości od obserwatora. Jest oczywiste, że kule leżące bliżej będą przesłaniać te znajdujące się dalej. Rysując kule ręcznie np. na tablicy, zaczęlibyśmy od tych położonych najgłębiej. Przed narysowaniem każdej z kul najpierw wymazywalibyśmy kolisty obszar, w którym miałby znaleźć się jej rzut. Gdyby obszar ten obejmował fragment obiektu już istniejącego na tablicy, znaczyłoby to, że fragment ten zostanie przesłonięty, więc jako niewidoczny także zostałby wymazany.

Podobnie postępować będziemy na ekranie. Zaczniemy od atomów położonych najdalej od obserwatora, a zatem cechujących się największą wartością współrzędnej Y (w przybliżeniu — uwzględniamy upraszczające założenia aksonometrii). Przed przystąpieniem do rysowania uporządkujemy więc tablice **X**, **Y**, **Z** i **R** tak, aby atomy ułożone były w kolejności malejącej współrzędnej Y. Obiekt o największej wartości **Y** będzie opisany jako nr 1 itd. Procedura sortująca atomy według wartości **Y** umieszczona jest w liniach 11Ø—19Ø. Zastosowano algorytm sortowania bąbelkowego. Nie jest on szczególnie efektywny, za to cechuje się prostotą i przejrzystością. Jego istota polega na tym, że wielokrotnie przeglądamy tablicę **Y**, porównując ze sobą pary sąsiednich elementów: 1 z 2, 2 i 3, 3 i 4 itd., aż do $n-1$ z n. W tablicy n-elementowej jest $n-1$ par elementów sąsiednich, stąd postać instrukcji 12Ø. Jeśli przy porównaniu elementów w danej parze (linia 13Ø) okaże się, że są one ułożone we właściwej kolejności (tutaj: pierwszy element jest nie mniejszy niż drugi), przestajemy interesować się tą parą, przechodząc do następnej. Jeśli jednak elementy pary ułożone są odwrotnie (mniejszy przed większym), to obydwa elementy należy zamienić miejscami. Nie wystarczy przy tym zamienić tylko elementy w tablicy **Y** — trzeba równocześnie w ten sam sposób przestawić zawartości odpowiednich komórek pozostałych tablic. Tylko w ten sposób zapewnimy sobie, że elementy wszystkich tablic z tym samym indeksem odnoszą się do tego samego punktu (atomu).

Przestawianie elementów w poszczególnych tablicach odbywa się w liniach 14Ø—17Ø. Po każdej operacji przestawiania zmienna **F** przyjmuje wartość 1. Zmienna ta jest zerowana przed każdym kolejnym przeglądaniem tablicy (linia 11Ø). Zmienna **F** zachowa wartość Ø po zakończeniu przeglądu wszystkich par tylko wtedy, gdy nie wykonano ani jednej zamiany. Jeśli nie było potrzeby wykonania żadnej zamiany, to znaczy, że nie wykryto ani jednej pary sąsiednich elementów, która byłaby źle uporządkowana. Jednym

słowem, zerowa wartość zmiennej **F** po przeglądzie tablicy jest wskaźnikiem zakończenia procesu sortowania. Jeśli F=1, przegląd połączony z ewentualnymi dalszymi zamianami trzeba powtórzyć.
Po zakończeniu sortowania przystępujemy wreszcie do rysowania kul-atomów. Realizuje to pętla 2ØØ—23Ø. Zmienne **Xe** i **Ye** otrzymują dzięki funkcjom **FN X** i **FN Y** wartości współrzędnych ekranowych środków poszczególnych kul, zmienna **Re** staje się równa ekranowemu promieniowi danej kuli. Właściwym rysowaniem kuli zajmuje się podprogram 1ØØØ. Zaczynamy jak na tablicy: "ścieramy" wnętrze kolistego obszaru o promieniu **Re**. Odbywa się to nieco po partyzancku, wykorzystujemy mianowicie instrukcję **CIRCLE INVERSE 1** i kreślimy koncentryczne okręgi, tyle tylko, że nie barwą atramentu, lecz barwą tła. Powstać może wątpliwość, co znaczy **STEP Ø.5** (linia 1ØØØ). Chodzi o możliwie dokładne oczyszczenie obszaru. Prosty eksperyment ze **STEP 1** wyjaśni, w czym rzecz.
Wokół oczyszczonego wnętrza zataczamy okrąg — kontur rzutu kuli (linia 1Ø1Ø). Teraz należy kulę tak wycieniować, aby wytworzyć złudzenie jej wypukłości. Najprościej osiągnąć to przez zaczernienie obszarów obrzeża, chociażby metodą przypadkowego rozrzucania kropek. Oglądane z dalszej odległości kropki zlewają się, stwarzając wrażenie jednolitej szarości. Poziom szarości jest wprost proporcjonalny do gęstości rozsianych kropek. Aby uniknąć uniformizacji kul, trzeba postarać się o pewną przypadkowość w rozmieszczeniu czarnych punktów. Przykładem praktycznej realizacji tego zadania jest pętla 1Ø2Ø—1Ø7Ø. Najpierw (linia 13Ø) losowany jest kąt, pod jakim ma być rzucony punkt względem środka okręgu. Następnie, w podobny sposób losowana jest odległość, jaka dzielić ma punkt od środka. O ile jednak prawdopodobieństwo wyboru dowolnego kąta z przedziału Ø—2Pi jest takie samo (przynajmniej teoretycznie), to z odległością jest inaczej (linia 1Ø4Ø). Postać wyrażenia gwarantuje, że promień nigdy nie będzie mniejszy niż ok. 4Ø% promienia okręgu, a prawdopodobieństwo ułożenia punktu w okolicy konturu jest większe niż w wewnętrznym obszarze koła. Teraz można już przeliczyć promień i kąt na odpowiednie współrzędne ekranowe (linia 1Ø5Ø) i nanieść punkt na ekran. Gęstość cieniowania jest uzależniona od ilości powtórzeń pętli (linia 1Ø2Ø). Zamiast stałej 15Ø można wstawić tam np. zmienny współczynnik, uzależniony np. od **Re**. Możliwości ulepszeń jest sporo. Program 80, podobnie jak większość innych programów w tej książce, stanowić ma tylko inspirację do własnych prac programistycznych.
Po wyświetleniu układu atomów w zadanej projekcji istnieje możliwość obrotów wokół każdej z trzech osi XYZ. Program pyta o kąty obrotu i transformuje współrzędne w sposób identyczny, jak w programie 78b.
Dane, zawarte w zbiorze **DATA** programu 80, opisują cząsteczkę bisphenolu. Nie zastosowano zróżnicowanych promieni ani cieniowania — otwiera się tu więc pole do własnych eksperymentów. Niektóre ujęcia cząsteczki pod różnymi kątami, uzyskane za pomocą programu 80, ukazuje rys. 12.

Prosta gra komputerowa

Większość użytkowników mikrokomputera pierwszy raz zetknęła się z maszyną za pośrednictwem gier. Zaprogramowanie dobrej gry z efektami pseudoprzestrzennymi i animacją wymaga z reguły dużych umiejętności programistycznych, zwłaszcza w języku maszynowym. Niekiedy jednak można zrealizować ciekawą grę dość prostymi środkami, nawet w języku BASIC. Decydujący jest pomysł i rozsądny wybór założeń, które powinny uwzględniać możliwości mikrokomputera. Przykładem może być poniższy program, symulujący na ekranie grę w tenisa ziemnego. Na ekranie ukazuje się widziany perspektywicznie kort tenisowy. Zawodnik grający w głębi sterowany jest automatycznie przez komputer, a zawodnik widoczny u dołu ekranu sterowany jest przez gracza. Sterowanie polega na takim przemieszczaniu zawodnika w lewo lub w prawo za pomocą klawiszów "5" i "8", by znalazł się on na trasie lotu piłki. Serwis następuje automatycznie, zawsze po stronie zawodnika, który ostatnio stracił punkt. Sposób obliczania punktów przypomina tenis ziemny: mecz składa się z jednego seta, podzielonego z kolei na gemy. Gem wygrywa ten zawodnik, który pierwszy przekroczył poziom 40 punktów (wygrał czwartą piłkę w gemie). Nie ma więc gry do przewagi dwóch wygranych piłek. Podobnie jest w secie, który kończy się w chwili wygrania przez któregoś z zawodników szóstego gema. Animację, polegającą na przedstawieniu ruchu piłki i przemieszczeń obydwu zawodników, osiągnięto dzięki zastosowaniu UDG oraz trybu **OVER 1**.

Dane o kształcie czterech zastosowanych UDG umieszczone są w zbiorze **DATA** (linie 42Ø i 43Ø). Zaraz po uruchomieniu programu dane przenoszone są za pomocą pętli (linia 2Ø) do pól opisujących UDG "A", "B", "C", "D". "A" przedstawia sylwetkę zawodnika "komputerowego", "B" — głowę i tors, "C" zaś — nogi zawodnika sterowanego przez grającego. UDG "D" obrazuje piłkę.

W liniach 5Ø—6Ø kreślone jest boisko. Jego kształt zakodowany jest w liniach **DATA** 44Ø—46Ø, w postaci par współrzędnych opisujących poszczególne punkty charakterystyczne (narożniki) kortu. Koniec listy współrzędnych sygnalizuje "wartownik" — X=999. Jeśli współrzędna X jest mniejsza od 999, lecz przekracza 299, oznacza to, że dany punkt rozpoczyna nowy kontur i nie należy go łączyć z poprzednikiem. Rzeczywistą wartość X uzyskuje się po odjęciu od wartości odczytanej stałej 5ØØ.

Program 81

```
10 REM TENIS ZIEMNY NA ZX SPECTRUM W JEZYKU BASIC
20 FOR A=USR "A" TO USR "D"+7: READ X: POKE A,X: NEXT A
30 INPUT "Stopien trudnosci (0-3) ";ST: LET ST=(ST+2)/5
40 PAPER 4: BORDER 4: INK 7: CLS
50 READ X,Y: IF X<300 THEN DRAW X,Y: GO TO 50
60 IF X<999 THEN PLOT X-500,Y: GO TO 50
70 FOR I=72 TO 84 STEP 2: PLOT 70,I: DRAW 115,0: NEXT I
80 FOR I=75 TO 180 STEP 2: PLOT I,70: DRAW 0,15: NEXT I
```

```
 90 READ A,B,A1,B1,X,Y,X0,Y0,DX,DY,W1,W2,S1,S2
100 PRINT AT X,Y; OVER 1;"d": GO SUB 380: GO TO 130

110 IF A=A1 AND B=B1 THEN GO TO 140
120 PRINT AT 8,A1; INK 2; OVER 1;"a";AT 19,B1; INK 7;"b";AT 20,B1;"c"
130 PRINT AT 8,INT A; INK 2; OVER 1;"a";AT 19,B; INK 1;"b";AT 20,B;"c"
140 PRINT AT INT X,INT Y; OVER 1;"d";AT X0,Y0; OVER 1;"d"
150 LET X0=INT X: LET Y0=INT Y: LET A1=INT A: LET B1=B
160 IF x0=8 THEN GO TO 210
170 IF X0=19 THEN GO TO 280
180 LET B=B+(INKEY$="8" AND B<26)-(INKEY$="5" AND B>5)
190 LET A=A+ST*((A1<INT Y AND A<20)-(A1>INT Y AND A>11))
200 LET X=X+DX: LET Y=Y+DY: GO TO 110

210 IF INT Y<>INT A THEN GO TO 250
220 BEEP .06,0
230 LET R=RND: LET DY=(R>.6 AND A<19)/2-(R<.4 AND A>12)/2
240 LET X=X+1: LET DX=1: GO TO 110
250 BEEP .2,-20: LET W2=W2+1
260 IF W2>3 THEN LET S2=S2+1: LET W2=0: LET W1=0: IF S2>5 THEN GO TO 340
270 FOR T=1 TO 99: NEXT T: LET Y=A: GO SUB 380: GO TO 220

280 IF INT Y<>B THEN GO TO 310
290 BEEP .05,20
300 LET DY=(B<15)/2-(B>17)/2: LET DX=-1: LET X=X-1: GO TO 110
310 BEEP .2,-20: LET W1=W1+1
320 IF W1>3 THEN LET S1=S1+1: LET W1=0: LET W2=0: IF S1>5 THEN GO TO 340
330 FOR T=1 TO 99: NEXT T: LET Y=B: GO SUB 380: GO TO 290

340 PRINT AT 14,3; INK 0; PAPER 7;
350 IF S2=6 THEN PRINT "Wygrales mecz.  Wynik: ";S1;":";S2: BEEP 1,-9
360 IF S1=6 THEN PRINT "Przegrales mecz.Wynik: ";S1;":";S2: BEEP 1,9
370 PRINT AT 16,4; FLASH 1;"sprobuj zagrac ponownie!": PAUSE 0: RUN

380 PRINT AT 0,0; INK 2; PAPER 7;S1;"  "; BRIGHT 1;FN W(W1);" "
390 PRINT #0;AT 1,0; INK 1; PAPER 7;S2;"  "; BRIGHT 1;FN W(W2);" "
400 RETURN
410 DEF FN W(P)=15*P-5*(P=3)
420 DATA 3,27,25,62,88,24,36,66,3,3,63,61,61,1,127,127,188,188
430 DATA 60,60,102,102,102,231,0,0,24,60,60,24,0,0
440 DATA 540,16,175,0,-47,87,-81,0,-47,-87,556,16,39,87,699,16
450 DATA -39,87,570,45,115,0,627,45,0,25,590,90,75,0,627,90,0,-7
460 DATA 670,70,115,0,0,15,-115,0,0,-15,999,0
470 DATA 15,15,15,15,18,15,18,15,-1,0,0,0,0,0
```

Instrukcje 7Ø i 8Ø rysują w środku kortu siatkę, a w linii 9Ø nadawane są zmiennym wartości początkowe. Także i one odczytywane są ze zbioru **DATA** (linia 47Ø).

Zmienne **A** i **B** przedstawiają aktualne położenie poziome odpowiednio zawodnika "elektronicznego" i sterowanego przez grającego (współrzędne pionowe są niezmienne). Zmienne **A1** i **B1** przedstawiają poprzednie położenia obydwu zawodników.

Zmienne **X** i **Y** przedstawiają aktualne współrzędne piłki, a **XØ** i **YØ** — położenie poprzednie. Zmienne **DX** i **DY** decydują o sposobie przemieszczania się piłki i przedstawiają przyrosty odpowiednich współrzędnych od kroku do kroku. Zmienne **W1** i **W2** przedstawiają liczbę piłek wygranych w gemie odpowiednio przez komputer i grającego, zmienne **S1** i **S2** to liczba gemów wygranych przez obydwie strony od początku seta.

Właściwa część programu — jego tzw. pętla główna — to linie 11Ø—2ØØ.

Uwaga! Przy wprowadzaniu linii 1ØØ oraz 12Ø—14Ø wszędzie tam, gdzie w instrukcjach **PRINT** jako stałe łańcuchowe występują **małe litery "a"—"d"**, należy wcisnąć odpowiedni klawisz w **trybie graficznym**, aby spowodować wywołanie odpowiedniego **UDG**.

Aby przemieścić któregoś z zawodników w nowe miejsce, trzeba najpierw skasować jego obraz w starym miejscu (linia 12Ø), a następnie wpisać sylwetkę w nowe (linia 13Ø). Podobnie przemieszczana jest piłka. W liniach 16Ø i 17Ø sprawdza się, czy piłka nie osiągnęła przypadkiem którejś z linii końcowych. Jeśli tak, następuje skok do odpowiedniej procedury sprawdzającej, czy w tym samym miejscu znajduje się zawodnik. Jeśli tak, przyjmuje się, że piłka została odbita. W przeciwnym razie przeciwnej stronie zostanie przyznany kolejny punkt.

Po każdym odbiciu piłki przez komputer następuje wyznaczanie nowych parametrów jej toru z wykorzystaniem funkcji **RND** (linia 23Ø). W przypadku odbicia piłki przez gracza sterowanego "ręcznie" tor piłki jest wyznaczany na podstawie aktualnego położenia zawodnika (linia 3ØØ). Po zmianie stanu meczu wskutek wygrania piłki przez którąś ze stron, wyświetlany jest na ekranie aktualny rezultat. W celu przeliczania liczby wygranych piłek na tenisowe punkty zdefiniowano funkcję **FN W** (linia 41Ø). Wyświetleniem wyniku zajmuje się podprogram 38Ø.

Odbiciom piłek przez zawodników, a także wyjściu piłki poza boisko, towarzyszą odpowiednie efekty dźwiękowe, zaprogramowane przy użyciu instrukcji **BEEP**. Jeśli któraś ze stron uzyskała szóstego gema, stan ten jest wykrywany w liniach 35Ø, 36Ø i wyświetlany jest odpowiedni komunikat wraz z wynikiem końcowym, grający zapraszany jest do kolejnego meczu.

Po utracie piłki, a przed kolejnym serwisem, niezbędna jest krótka przerwa. Osiągnięto ją przez pętlę opóźniającą, wykorzystującą **FOR... NEXT**, linie 27Ø i 33Ø. Pozornie prostsze byłoby zastosowanie instrukcji **PAUSE**. Pamiętajmy jednak, że powodowana przez tę instrukcję przerwa w wykonywaniu programu kończy się bezwarunkowo w chwili wciśnięcia jakiegokolwiek klawisza. Jest rzeczą bardzo prawdopodobną, że grający mimo przerwy w grze będzie wciskał ciągle jakiś klawisz, skracając w nieprzewidziany sposób pauzę przed serwisem.

Sterowanie zawodnikiem dolnym odbywa się w linii 18Ø, która bada klawiaturę funkcją **INKEY$** i rozpoznaje wciśnięcie któregoś z dwóch klawiszy kierunkowych "5" lub "8" (można oczywiście wybrać inne klawisze). Sterowanie zawodnikiem "komputerowym" ma miejsce w linii 19Ø. Nowe położenie tego zawodnika wyznaczane jest na podstawie tego, z której jego strony znajduje się aktualnie piłka. Używając terminów z dziedziny automatyki, powiedzielibyśmy, że mamy do czynienia z tzw. regulacją nadążną. Stopień trudności określa podatność zawodnika na taką regulację: im wyższy stopień trudności **ST**, tym żwawiej zawodnik jest w stanie reagować na przemieszczenia piłki.

Przedstawiony program jest zaledwie szkieletem gry komputerowej. Można

go znacznie ulepszyć, wprowadzając np. rysunek trybun, obliczanie stanu meczu ściśle zgodne z tenisowymi przepisami (przewaga dwóch piłek niezbędna do wygrania gema itp.). Zastosowane w grze rozwiązania są dość typowe dla wielu innych gier pisanych w języku BASIC i nie tylko. Dotyczy to zwłaszcza wykorzystania instrukcji **PRINT OVER 1** do wywoływania wrażenia ruchu na ekranie oraz sposobów modelowania wydarzeń w miniaturowym świecie działającego programu. Powyższy program zainspirowany został artykułem Gerda Hubera w miesięczniku "Happy Computer".
Można zapytać, czy warto żmudnie wprowadzać z klawiatury listing gry w języku BASIC, tym bardziej że zapewne trzeba będzie jeszcze wytropić błędy. Jeśli zależy nam tylko na ludystycznym zadowoleniu, to z pewnością nie warto. Gotowe programy gier, stworzone przez profesjonalistów kosztem znacznych nakładów czasu i wysiłku, zachwycają techniczną finezją.
Sama gra nudzi się szybko. Nie można powiedzieć tego o pasjonującym zajęciu, jakim jest samodzielne układanie programów. Komputer pozwala nam na swobodną realizację najdziwniejszych nawet pomysłów, modelowanie wycinków rzeczywistości albo tworzenie nie znanych do tej pory światów. Jest to szczególną wartością dla osób szukających okazji do twórczego wyżycia się. O tym, że komputer może być wspaniałym pomocnikiem w pracy i w nauce, nie trzeba wspominać.

Aby móc poznać prawdziy smak władzy nad komputerem, trzeba jednak w dostatecznym stopniu opanować podstawowe elementy programistycznego rzemiosła. Inaczej niedostatek praktycznych, "narzędziowych" umiejętności będzie kulą u nogi w realizacji najciekawszych nawet i najoryginalniejszych pomysłów.
Nawet tak prosty komputer jak ZX Spectrum kryje w sobie nie zawsze doceniane możliwości. Zadaniem niniejszej książki było tylko je zasygnalizować. Niech będzie ona zachętą do dalszej, dociekliwej wędrówki po świecie mikroinformatyki.

Spis treści

Cena zł 180,—